

UNIVERSITY OF ARIZONA



39001034412208

# Handbook of Logic in Computer Science

## Volume 3 Semantic Structures

Edited by  
**S. ABRAMSKY, DOV M. GABBAY,**  
**and T. S. E. MAIBAUM**

OXFORD SCIENCE PUBLICATIONS









Digitized by the Internet Archive  
in 2025



# HANDBOOK OF LOGIC IN COMPUTER SCIENCE

---

Editors

S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum

HANDBOOKS OF LOGIC IN COMPUTER SCIENCE  
*and*  
ARTIFICIAL INTELLIGENCE AND LOGIC PROGRAMMING

*Executive Editor*

Dov M. Gabbay

*Administrator*

Jane Spurr

---

Handbook of Logic in Computer Science

- Volume 1** Background: Mathematical structures
- Volume 2** Background: Computational structures
- Volume 3** Semantic structures
- Volume 4** Semantic modelling
- Volume 5** Theoretical methods in specification and verification
- Volume 6** Logical methods in computer science

Handbook of Logic in Artificial Intelligence and  
Logic Programming

- Volume 1** Logical foundations
- Volume 2** Deduction methodologies
- Volume 3** Nonmonotonic reasoning and uncertain reasoning
- Volume 4** Epistemic and temporal reasoning
- Volume 5** Logic programming

# Handbook of Logic in Computer Science

Volume 3  
Semantic Structures

Edited by

S. ABRAMSKY

*Professor of Computing Science*

DOV M. GABBAY

*Professor of Computing Science*

and

T. S. E. MAIBAUM

*Professor of Foundations of  
Software Engineering*

*Imperial College of Science, Technology and Medicine  
London*

Volume Co-ordinator

S. ABRAMSKY

CLARENDON PRESS · OXFORD

1994



*Oxford University Press, Walton Street, Oxford OX2 6DP*

*Oxford New York*

*Athens Auckland Bangkok Bombay*

*Calcutta Cape Town Dar es Salaam Delhi*

*Florence Hong Kong Istanbul Karachi*

*Kuala Lumpur Madras Madrid Melbourne*

*Mexico City Nairobi Paris Singapore*

*Taipei Tokyo Toronto*

*and associated companies in*

*Berlin Ibadan*

*Oxford is a trade mark of Oxford University Press*

*Published in the United States by*

*Oxford University Press Inc., New York*

*© The contributors listed on p. xv, 1994*

*All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior permission in writing of Oxford University Press. Within the UK, exceptions are allowed in respect of any fair dealing for the purpose of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act, 1988, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms and in other countries should be sent to the Rights Department, Oxford University Press, at the address above.*

*This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out, or otherwise circulated without the publisher's prior consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser.*

*A catalogue record for this book is available from the British Library*

*Library of Congress Cataloging in Publication Data*

*(Data available)*

*ISBN 0 19 853762 X*

*Typeset using LaTeX by Jane Spurr*

*Printed in Great Britain by*

*Biddles Ltd*

*Guildford and King's Lynn*

# Preface

We are happy to present Volume 3 of our Handbook project, on *Semantic Structures*. The previous two volumes presented the background on fundamental mathematical structures—consequence relations, model theory, recursion theory, category theory, universal algebra, topology, and on computational structures—term-rewriting systems,  $\lambda$ -calculi, modal and temporal logics and algorithmic proof systems. The computational structures considered thus far have been predominantly syntactic in character; while the discussion of mathematical structures has been quite general and free-standing.

In the present volume these threads are drawn together. We look at how mathematical structures are used to model computational processes. At first sight, this enterprise looks quite similar to the standard notions of model theory in logic. However, programming language semantics has significant special features and problems of its own, which have required an extensive development both of novel semantic techniques, and of the underlying mathematical foundations.

The first chapter of this volume concerns Domain Theory, a mathematical theory originated by Dana Scott to provide a mathematical foundation for the semantics of programming languages. In particular, Domain theory allows meaning to be given to *recursive definitions*, both of programs and of data types. In the 25 years since its inception, a very rich mathematical theory has been developed, and the chapter gives a systematic presentation of this theory.

The second Chapter concerns denotational semantics. The idea here is to assign meaning to programs as elements of, or functions between, suitable mathematical structures. Crucially, this assignment of meanings should be *compositional*; the meaning  $\llbracket C(P_1, \dots, P_n) \rrbracket$  of a complex program formed by applying some constructor  $C$  to the sub-programs  $P_1, \dots, P_n$  should be a function of the meaning of its parts:

$$\llbracket C(P_1, \dots, P_n) \rrbracket = C(\llbracket P_1 \rrbracket, \dots, \llbracket P_n \rrbracket),$$

where  $C$  is a suitable mathematical operation on meanings. In practice, the “suitable mathematical structures” will very often be domains; note also that the above equation can be read as saying that semantics is a *homomorphism*, thus invoking concepts of universal algebra. Chapter 2 on



denotational semantics sets out the general principles and techniques of compositional semantics, and shows how they can be applied to a range of programming language features.

Chapter 3 takes up the algebraic theme, focussing on how syntax can be viewed as a free algebra, and a semantics as another algebra, with the semantic algebra as a homomorphism between them (uniquely given because of the freeness of the syntactic algebra). For this programme to work properly it is necessary to combine ideas from Universal algebra (Volume 1) and Domain theory; ideas from category theory are also used extensively.

Finally, Chapter 4 concerns the semantics of types. Types are of increasing importance in modern programming languages. They provide a very effective conceptual discipline in program design, supported by efficient type-checking algorithms which allow many program errors to be detected by the compiler. A wide variety of type systems for programming languages have been considered, embracing such notions as higher-order and recursive types, polymorphism and subtyping. Chapter 4 provides an overview of such type systems and their semantics, which plays an essential role e.g. in establishing the soundness of type inference systems. For the very rich type systems currently being studied, many foundational problems are raised by the work on semantics, so this area shows an important interplay between theory and practice.

## The Handbooks

The *Handbook of Logic in Theoretical Computer Science* and its companion, the *Handbook of Logic in Artificial Intelligence and Logic Programming*, have been created in response to a growing need for an in-depth survey of the application of logic in computer science and AI.

We see the creation of the Handbook as a combination of authoritative exposition, comprehensive survey, and fundamental research exploring the underlying unifying themes in the various areas. The intended audience is graduate students and researchers in the areas of computing and logic, as well as other people interested in the subject. We assume as background some mathematical sophistication. Much of the material will also be of interest to logicians and mathematicians.

The tables of contents of the volumes were finalized after extensive discussions between Handbook authors and second readers. The first two volumes present the Background—Mathematical Structures and Computational Structures.

The chapters, which in many cases are of monographic length and scope, are written with emphasis on possible unifying themes. The chapters have an overview, introduction, and main body. A final part is dedicated to more specialized topics.

Chapters are written by internationally renowned researchers in their



respective areas. The chapters are co-ordinated and their contents were discussed in joint meetings. Each chapter has been written using the following procedures:

1. A very detailed table of contents was discussed and co-ordinated at several meetings between authors and editors of related chapters. The discussion was in the form of a series of lectures by the authors. Once an agreement was reached on the detailed table of contents, the authors wrote a draft and sent it to the editors and to other related authors. For each chapter there is a second reader (the first reader is the author) whose job it has been to scrutinize the chapter together with the editors. The second reader's role is very important and has required effort and serious involvement with the authors.

Second readers for this volume are:

Chapter 1: Domain Theory—R Heckman

Chapter 2: Denotational Semantics—C Wadsworth

Chapter 3: Algebraic Semantics—I Guessarian

Chapter 4: Semantics of Types—R Crole

2. Once this process was completed (i.e. drafts seen and read by a large enough group of authors), there were other meetings on several chapters in which authors lectured on their chapters and faced the criticism of the editors and audience. The final drafts were prepared after these meetings.
3. We attached great importance to group effort and co-ordination in the writing of chapters. The first two parts of each chapter, namely the introduction-overview and main body, are not completely under the discretion of the author, as he/she had to face the general criticism of all the other authors. Only the third part of the chapter is entirely for the authors' own personal contribution.

The Handbook meetings were generously financed by OUP, by SERC contract SO/809/86, by the Department of Computing at Imperial College, and by several anonymous private donations.

We would like to thank our colleagues, authors, second readers, and students for their effort and professionalism in producing the manuscripts for the Handbook. We would particularly like to thank the staff of OUP for their continued and enthusiastic support, and Mrs Jane Spurr, our OUP Administrator, for her dedication and efficiency.

London  
April 1994

S. Abramsky and D. M. Gabbay



# Contents

## List of contributors

xv

## Domain theory

*Samson Abramsky and Achim Jung*

1	Introduction and Overview	2
1.1	Origins	2
1.2	Our approach	4
1.3	Overview	5
2	Domains individually	6
2.1	Convergence	6
2.2	Approximation	15
2.3	Topology	27
3	Domains collectively	32
3.1	Comparing domains	32
3.2	Finitary constructions	39
3.3	Infinitary constructions	44
4	Cartesian closed categories of domains	52
4.1	Local uniqueness: Lattice-like domains	54
4.2	Finite choice: Compact domains	55
4.3	The hierarchy of categories of domains	62
5	Recursive domain equations	66
5.1	Examples	67
5.2	Construction of solutions	69
5.3	Canonicity	74
5.4	Analysis of solutions	79
6	Equational theories	84
6.1	General techniques	85
6.2	Powderdomains	94
7	Domains and logic	107
7.1	Stone duality	108
7.2	Some equivalences	114
7.3	The logical viewpoint	124
8	Further directions	148
8.1	Further topics in ‘classical domain theory’	148
8.2	Stability and sequentiality	151
8.3	Reformulations of domain theory	152



8.4	Axiomatic domain theory	154
8.5	Synthetic domain theory	155
9	Guide to the literature	156

## **Denotational semantics** 169

*R. D. Tennent*

1	Introduction	170
1.1	Approaches	171
1.2	An example: binary numerals	172
1.3	Compositionality	175
1.4	Criteria	176
1.5	Overview	178
1.6	Bibliographic notes	179
2	A simple imperative language	180
2.1	Expressions and commands	180
2.2	Assignment commands	185
2.3	Indefinite iterations	187
2.4	Programs	191
2.5	Operational semantics	192
2.6	Programming logic	196
2.7	Non-determinism	203
2.8	Bibliographic notes	205
3	A simple applicative language	205
3.1	Definitions and function applications	206
3.2	Function definitions	212
3.3	Defined notation	214
3.4	Elementary properties	217
3.5	Programming logic	220
3.6	Bibliographic notes	228
4	Recursion	228
4.1	Recursive definitions	228
4.2	Domain-theoretic semantics	231
4.3	Operational semantics	237
4.4	Programming logic	239
4.5	Full abstraction	242
4.6	Untyped procedures	243
4.7	Bibliographic notes	245
5	An Algol-like language I	245
5.1	Syntax	246
5.2	Semantics	250
5.3	Call by value	253
5.4	Programming logic	256

5.5 Bibliographic notes	260
6 An Algol-like language II	260
6.1 Coercions	261
6.2 Local variables	268
6.3 Product types and arrays	270
6.4 Lists	274
6.5 Acceptors	279
6.6 Jumps	282
6.7 Intermediate output	287
6.8 Block expressions	288
6.9 Bibliographic notes	289
7 Possible worlds	290
7.1 Functor–category semantics	291
7.2 Semantic-domain functors	292
7.3 Semantic valuations	295
7.4 Semantics of local variables	298
7.5 Specifications	302
7.6 Non-interference specifications	304
7.7 Semantics of block expressions	308
7.8 Bibliographic notes	311

## **Algebraic semantics** 323

*Eric G. Wagner*

1 Introduction and motivation	324
1.1 General remarks	324
1.2 Some motivating examples	326
1.3 A notational ‘crisis’	332
2 Algebraic theories, definitions and examples	332
2.1 General remarks	332
2.2 Basic definitions	332
2.3 Algebraic theories as categories	336
2.4 Examples of algebraic theories	337
2.5 Notations and basic identities	341
3 Ordered theories	343
3.1 Definition of ordered and continuous theories	343
3.2 Examples of ordered theories	344
4 Theories with iteration operators	348
4.1 Iteration operators	348
4.2 Iteration, rational, and iterative theories	349
4.3 The V-operator	356
4.4 Iteration operators and flowcharts	356

4.5	Interpretations of flowcharts	364
5	More about iteration operators	366
5.1	Relationships between iteration, rational, and iterative theories	366
5.2	Some identities for iteration operators	372
6	Iteration closure, and normal form theorems	374
7	Free theories and Herbrand interpretations	378
7.1	Free theories	379
7.2	The general case	380
8	Recursive hierarchies	386

## **The semantics of types in programming languages** 395

*Carl A. Gunter*

1	Introduction	396
2	Types in programming	397
2.1	Higher types	397
2.2	Recursive types	400
2.3	Parametric polymorphism	402
2.4	Subtypes	404
3	Simple types as sets	408
3.1	Types and equations	409
3.2	Sets as a model	412
3.3	Type frames	415
3.4	Completeness for sets	418
4	Simple types as domains	420
4.1	A programming language for computable functions	420
4.2	Operational semantics	422
4.3	Operational equivalence	425
4.4	bc-domains and dl-domains	426
4.5	Full abstraction	427
5	Types as invariants	430
5.1	Run-time safety	430
5.2	Implicit types	434
5.3	Run-time safety for assignments and continuations	441
6	Types as subsets	446
6.1	Untyped $\lambda$ -calculus	446
6.2	What is a model of the untyped $\lambda$ -calculus?	448
6.3	What models of the untyped $\lambda$ -calculus are there?	449
6.4	Inclusive subsets as types	451
6.5	Subtyping as subset inclusion	455
7	Types as partial equivalence relations	458
7.1	Sets as a model of $ML_0$ types	458



7.2	Another typing system for $ML_0$	460
7.3	The polymorphic $\lambda$ -calculus	461
7.4	Sets as a model of polymorphic types?	464
7.5	Simple types as PERs	466
7.6	PERs as a model of polymorphic types	468
8	Conclusion	470
<b>Index</b>		477



# Contributors

**Samson Abramsky**

Department of Computing, Imperial College of Science, Technology and Medicine, London SW7 2BZ, UK.

**Carl A. Gunter**

Department of Computer and Information Science, 200 South 33rd Street, University of Pennsylvania, Philadelphia PA 19104-6389, USA.

**Achim Jung**

Fachbereich Mathematik, Technische Hochschule Darmstadt, Schlossgartenstrasse 7, D-64289 Darmstadt, Germany.

**R. D. Tennent**

Queen's University, Kingston, Ontario K7L 3N6, Canada.

**Eric C. Wagner**

Mathematical Sciences Department, IBM, Thomas J. Watson Research Centre, PO Box 218, Yorktown Heights, New York 10598, USA.





# Domain Theory

Samson Abramsky and Achim Jung

---

## Contents

1	Introduction and Overview . . . . .	2
1.1	Origins . . . . .	2
1.2	Our approach . . . . .	4
1.3	Overview . . . . .	5
2	Domains individually . . . . .	6
2.1	Convergence . . . . .	6
2.2	Approximation . . . . .	15
2.3	Topology . . . . .	27
3	Domains collectively . . . . .	32
3.1	Comparing domains . . . . .	32
3.2	Finitary constructions . . . . .	39
3.3	Infinitary constructions . . . . .	44
4	Cartesian closed categories of domains . . . . .	52
4.1	Local uniqueness: Lattice-like domains . . . . .	54
4.2	Finite choice: Compact domains . . . . .	55
4.3	The hierarchy of categories of domains . . . . .	62
5	Recursive domain equations . . . . .	66
5.1	Examples . . . . .	67
5.2	Construction of solutions . . . . .	69
5.3	Canonicity . . . . .	74
5.4	Analysis of solutions . . . . .	79
6	Equational theories . . . . .	84
6.1	General techniques . . . . .	85
6.2	Powerdomains . . . . .	94
7	Domains and logic . . . . .	107
7.1	Stone duality . . . . .	108
7.2	Some equivalences . . . . .	114
7.3	The logical viewpoint . . . . .	124
8	Further directions . . . . .	148
8.1	Further topics in ‘classical domain theory’ . . . . .	148
8.2	Stability and sequentiality . . . . .	151

8.3	Reformulations of domain theory . . . . .	152
8.4	Axiomatic domain theory . . . . .	154
8.5	Synthetic domain theory . . . . .	155
9	Guide to the literature . . . . .	156

## 1 Introduction and Overview

### 1.1 Origins

Let us begin with the problems which gave rise to domain theory:

1. **Least fixpoints as meanings of recursive definitions.** Recursive definitions of procedures, data structures and other computational entities abound in programming languages. Indeed, recursion is the basic effective mechanism for describing infinite computational behaviour in finite terms. Given a recursive definition:

$$X = \dots X \dots, \quad (1.1)$$

how can we give a non-circular account of its meaning? Suppose we are working inside some mathematical structure  $D$ . We want to find an element  $d \in D$  such that substituting  $d$  for  $x$  in (1.1) yields a valid equation. The right-hand-side of (1.1) can be read as a function of  $X$ , semantically as  $f: D \rightarrow D$ . We can now see that we are asking for an element  $d \in D$  such that  $d = f(d)$ —that is, for a *fixpoint* of  $f$ . Moreover, we want a *uniform canonical* method for constructing such fixpoints for arbitrary structures  $D$  and functions  $f: D \rightarrow D$  within our framework. Elementary considerations show that the usual categories of mathematical structures either fail to meet this requirement at all (sets, topological spaces) or meet it in a trivial fashion (groups, vector spaces).

2. **Recursive domain equations.** Apart from recursive definitions of computational objects, programming languages also abound, explicitly or implicitly, in recursive definitions of *datatypes*. The classical example is the type-free  $\lambda$ -calculus [Barendregt, 1984]. To give a mathematical semantics for the  $\lambda$ -calculus is to find a mathematical structure  $D$  such that terms of the  $\lambda$ -calculus can be interpreted as elements of  $D$  in such a way that application in the calculus is interpreted by function application. Now consider the self-application term  $\lambda x.xx$ . By the usual condition for type-compatibility of a function with its argument, we see that if the second occurrence of  $x$  in  $xx$  has type  $D$ , and the whole term  $xx$  has type  $D$ , then the first occurrence must have, or be construable as having, type  $[D \rightarrow D]$ . Thus we are led to the requirement that we have

$$[D \rightarrow D] \cong D.$$

If we view  $[\cdot \rightarrow \cdot]$  as a functor  $F: \mathbf{C}^{op} \times \mathbf{C} \rightarrow \mathbf{C}$  over a suitable category  $\mathbf{C}$  of mathematical structures, then we are looking for a fixpoint  $D \cong F(D, D)$ . Thus recursive datatypes again lead to a requirement for fixpoints, but now lifted to the functorial level. Again we want such fixpoints to exist uniformly and canonically.

This second requirement is even further beyond the realms of ordinary mathematical experience than the first. Collectively, they call for a novel mathematical theory to serve as a foundation for the semantics of programming languages.

A first step towards domain theory is the familiar result that every monotone function on a complete lattice, or more generally on a directed-complete partial order with least element, has a least fixpoint. (For an account of the history of this result, see [Lassez *et al.*, 1982].) Some early uses of this result in the context of formal language theory were [Arden, 1960; Ginsburg and Rice, 1962]. It had also found applications in recursion theory [Kleene, 1952; Platek, 1964]. Its application to the semantics of first-order recursion equations and flowcharts was already well-established among computer scientists by the end of the 1960's [de Bakker and Scott, 1969; Bekič, 1969; Bekič, 1971; Park, 1969]. But domain theory proper, at least as we understand the term, began in 1969, and was unambiguously the creation of one man, Dana Scott [1969; 1970; 1971; 1972; 1993]. In particular, the following key insights can be identified in his work:

1. **Domains as types.** The fact that suitable categories of domains are *cartesian closed*, and hence give rise to models of typed  $\lambda$ -calculi. More generally, that domains give mathematical meaning to a broad class of data-structuring mechanisms.
2. **Recursive types.** Scott's key construction was a solution to the 'domain equation'

$$D \cong [D \rightarrow D]$$

thus giving the first mathematical model of the type-free  $\lambda$ -calculus. This led to a general theory of solutions of recursive domain equations. In conjunction with (1), this showed that domains form a suitable universe for the semantics of programming languages. In this way, Scott provided a mathematical foundation for the work of Christopher Strachey on denotational semantics [Milne and Strachey, 1976; Stoy, 1977]. This combination of descriptive richness and a powerful and elegant mathematical theory led to denotational semantics becoming a dominant paradigm in theoretical computer science.

3. **Continuity vs. computability.** *Continuity* is a central pillar of domain theory. It serves as a qualitative approximation to computability. In other words, for most purposes to detect whether some construction is computationally feasible it is sufficient to check that it is continuous, although continuity is an 'algebraic' condition, which is

much easier to handle than computability. In order to give this idea of continuity as a smoothed-out version of computability substance, it is not sufficient to work only with a notion of ‘completeness’ or ‘convergence’; one also needs a notion of *approximation*, which does justice to the idea that infinite objects are given in some coherent way as limits of their finite approximations. This leads to considering, not arbitrary complete partial orders, but the *continuous* ones. Indeed, Scott’s early work on domain theory was seminal to the subsequent extensive development of the theory of continuous lattices, which also drew heavily on ideas from topology, analysis, topological algebra and category theory [Gierz *et al.*, 1980].

4. **Partial information.** A natural concomitant of the notion of approximation in domains is that they form the basis of a theory of partial information, which extends the familiar notion of partial function to encompass a whole spectrum of ‘degrees of definedness’. This has important applications to the semantics of programming languages, where such multiple degrees of definition play a key role in the analysis of computational notions such as lazy vs. eager evaluation, and call-by-name vs. call-by-value parameter-passing mechanisms for procedures.

General considerations from recursion theory dictate that partial functions are unavoidable in any discussion of computability. domain theory provides an appropriately abstract, structural setting in which these notions can be lifted to higher types, recursive types, etc.

## 1.2 Our approach

It is a striking fact that, although domain theory has been around for a quarter-century, no book-length treatment of it has yet been published. Quite a number of books on semantics of programming languages, incorporating substantial introductions to domain theory as a necessary tool for denotational semantics, have appeared [Stoy, 1977; Schmidt, 1986; Gunter, 1992b; Winskel, 1993]; but there has been no text devoted to the underlying mathematical theory of domains. To make an analogy, it is as if many calculus textbooks were available, offering presentations of some basic analysis interleaved with its applications in modelling physical and geometrical problems; but no textbook of real analysis. Although this Handbook chapter cannot offer the comprehensive coverage of a full-length textbook, it is nevertheless written in the spirit of a presentation of real analysis. That is, we attempt to give a crisp, efficient presentation of the mathematical theory of domains without excursions into applications. We hope that such an account will be found useful by readers wishing to acquire some familiarity with domain theory, including those who seek to apply it. Indeed, we believe that the chances for exciting new applications of domain theory will be enhanced if more people become aware of the full



richness of the mathematical theory.

### 1.3 Overview

#### Domains individually

We begin by developing the basic mathematical language of domain theory, and then present the central pillars of the theory: convergence and approximation. We put considerable emphasis on bases of continuous domains, and show how the theory can be developed in terms of these. We also give a first presentation of the topological view of domain theory, which will be a recurring theme.

#### Domains collectively

We study special classes of maps which play a key role in domain theory: retractions, adjunctions, embeddings and projections. We also look at construction on domains such as products, function spaces, sums and lifting; and at bilimits of directed systems of domains and embeddings.

#### Cartesian closed categories of domains

A particularly important requirement on categories of domains is that they should be cartesian closed (i.e. closed under function spaces). This creates a tension with the requirement for a good theory of approximation for domains, since neither the category **CONT** of all continuous domains, nor the category **ALG** of all algebraic domains is cartesian closed. This leads to a non-trivial analysis of necessary and sufficient conditions on domains to ensure closure under function spaces, and striking results on the classification of the maximal cartesian closed full subcategories of **CONT** and **ALG**. This material is based on Jung [1989; 1990].

#### Recursive domain equations

The theory of recursive domain equations is presented. Although this material formed the very starting point of domain theory, a full clarification of just what canonicity of solutions means, and how it can be translated into proof principles for reasoning about these canonical solutions, has only emerged over the past two or three years, through the work of Peter Freyd and Andrew Pitts [Freyd, 1991; Freyd, 1992; Pitts, 1993a]. We make extensive use of their insights in our presentation.

#### Equational theories

We present a general theory of the construction of free algebras for inequational theories over continuous domains. These results, and the underlying constructions in terms of bases, appear to be new. We then apply this general theory to powerdomains and give a comprehensive treatment of the Plotkin, Hoare and Smyth powerdomains. In addition to characterizing these as free algebras for certain inequational theories, we also prove repre-



sentation theorems which characterize a powerdomain over  $D$  as a certain space of subsets of  $D$ ; these results make considerable use of topological methods.

## Domains and logic

We develop the logical point of view of domain theory, in which domains are characterized in terms of their observable properties, and functions in terms of their actions on these properties. The general framework for this is provided by Stone duality; we develop the rudiments of Stone duality in some generality, and then specialize it to domains. Finally, we present ‘domain theory in logical form’ [Abramsky, 1991b], in which a metalanguage of types and terms suitable for denotational semantics is extended with a language of properties, and presented axiomatically as a programming logic in such a way that the lattice of properties over each type is the Stone dual of the domain denoted by that type, and the prime filter of properties of a term which can be proved to hold correspond under Stone duality to the domain element denoted by that term. This yields a systematic way of moving back and forth between the logical and denotational descriptions of some computational situation, each determining the other up to isomorphism.

## Acknowledgements

We would like to thank Jiří Adámek, Reinhold Heckmann, Michael Huth, Mathias Kegelmann, Philipp Sünderhauf, and Paul Taylor for very careful proof reading. Achim Jung would particularly like to thank the people from the ‘Domain Theory Group’ at Darmstadt, who provided a stimulating and supportive environment.

Our major intellectual debts, inevitably, are to Dana Scott and Gordon Plotkin. The more we learn about domain theory, the more we appreciate the depth of their insights.

## 2 Domains individually

We will begin by introducing the basic language of Domain Theory. Most topics we deal with in this section are treated more thoroughly and at a more leisurely pace in [Davey and Priestley, 1990].

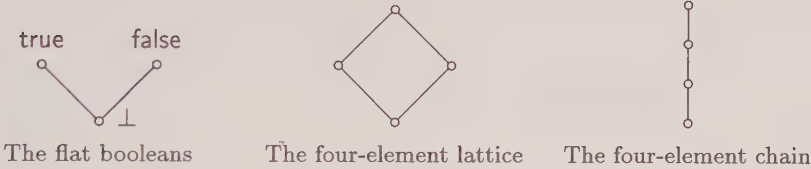
### 2.1 Convergence

#### 2.1.1 Posets and preorders

**Definition 2.1.1.** A set  $P$  with a binary relation  $\sqsubseteq$  is called a *partially ordered set* or *poset* if the following holds for all  $x, y, z \in P$ :

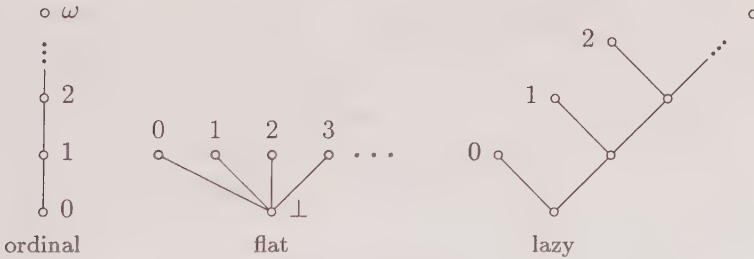
1.  $x \sqsubseteq x$  (Reflexivity)
2.  $x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$  (Transitivity)
3.  $x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$  (Antisymmetry)

Small finite partially ordered sets can be drawn as line diagrams (Hasse diagrams). Examples are given in Figure 1. We will also allow ourselves to



**Fig. 1.** A few posets drawn as line diagrams.

draw infinite posets by showing a finite part which illustrates the building principle. Three examples are given in Figure 2. We prefer the notation  $\sqsubseteq$

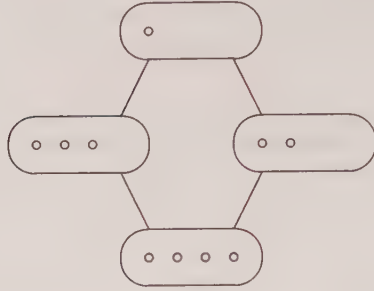


**Fig. 2.** Three versions of the natural numbers.

to the more common  $\leq$  because the order on domains we are studying here often coexists with an otherwise unrelated intrinsic order. The flat and lazy natural numbers from Figure 2 illustrate this.

If we drop antisymmetry from our list of requirements then we get what is known as *pre-orders*. This does not change the theory very much. As is easily seen, the sub-relation  $\sqsubseteq \cap \supseteq$  is in any case an equivalence relation and if two elements from two equivalence classes  $x \in A, y \in B$  are related by  $\sqsubseteq$ , then so is any pair of elements from  $A$  and  $B$ . We can therefore pass from a pre-order to a canonical partially ordered set by taking equivalence classes. Pictorially, the situation then looks as in Figure 3.

Many notions from the theory of ordered sets make sense even if reflexivity fails. Hence we may sum up these considerations with the slogan: *Order theory is the study of transitive relations*. A common way to extract



**Fig. 3.** A pre-order whose canonical quotient is the four-element lattice.

the order-theoretic content from a relation  $R$  is to pass to the transitive closure of  $R$ , defined as  $\bigcup_{n \in \mathbb{N} \setminus \{0\}} R^n$ .

Ordered sets can be turned upside down:

**Proposition 2.1.2.** *If  $\langle P, \sqsubseteq \rangle$  is an ordered set then so is  $P^{op} = \langle P, \sqsupseteq \rangle$ .*

One consequence of this observation is that each of the concepts introduced below has a dual counterpart.

### 2.1.2 Notation from order theory

The following concepts form the core language of order theory.

**Definition 2.1.3.** Let  $(P, \sqsubseteq)$  be an ordered set.

1. A subset  $A$  of  $P$  is an *upper set* if  $x \in A$  implies  $y \in A$  for all  $y \sqsupseteq x$ . We denote by  $\uparrow A$  the set of all elements above some element of  $A$ . If no confusion is to be feared then we abbreviate  $\uparrow\{x\}$  as  $\uparrow x$ . The dual notions are *lower set* and  $\downarrow A$ .
2. An element  $x \in P$  is called an *upper bound* for a subset  $A \subseteq P$ , if  $x$  is above every element of  $A$ . We often write  $A \sqsubseteq x$  in this situation. We denote by  $\text{ub}(A)$  the set of all upper bounds of  $A$ . Dually,  $\text{lb}(A)$  denotes the set of lower bounds of  $A$ .
3. An element  $x \in P$  is *maximal* if there is no other element of  $P$  above it:  $\uparrow x \cap P = \{x\}$ . Minimal elements are defined dually. For a subset  $A \subseteq P$  the minimal elements of  $\text{ub}(A)$  are called *minimal upper bounds* of  $A$ . The set of all minimal upper bounds of  $A$  is denoted by  $\text{mub}(A)$ .
4. If all elements of  $P$  are below a single element  $x \in P$ , then  $x$  is said to be the *largest element*. The dually defined *least element* of a poset is also called *bottom* and is commonly denoted by  $\perp$ . In the presence of a least element we speak of a *pointed poset*.
5. If for a subset  $A \subseteq P$  the set of upper bounds has a least element  $x$ , then  $x$  is called the *supremum* or *join*. We write  $x = \bigsqcup A$  in this

case. In the other direction we speak of *infimum* or *meet* and write  $x = \prod A$ .

6. A partially ordered set  $P$  is a  $\sqcup$ -*semilattice* ( $\sqcap$ -*semilattice*) if the supremum (infimum) for each pair of elements exists. If  $P$  is both a  $\sqcup$ - and a  $\sqcap$ -semilattice then  $P$  is called a *lattice*. A lattice is *complete* if suprema and infima exist for all subsets.

The operations of forming suprema, resp. infima, have a few basic properties which we will use throughout this text without mentioning them further.

**Proposition 2.1.4.** *Let  $P$  be a poset such that the suprema and infima occurring in the following formulae exist. ( $A, B$  and all  $A_i$  are subsets of  $P$ .)*

1.  $A \subseteq B$  implies  $\sqcup A \subseteq \sqcup B$  and  $\sqcap A \supseteq \sqcap B$ .
2.  $\sqcup A = \sqcup(\downarrow A)$  and  $\sqcap A = \sqcap(\uparrow A)$ .
3. If  $A = \bigcup_{i \in I} A_i$  then  $\sqcup A = \sqcup_{i \in I}(\sqcup A_i)$  and similarly for the infimum.

**Proof.** We illustrate order-theoretic reasoning with suprema by showing (3). The element  $\sqcup A$  is above each element  $\sqcup A_i$  by (1), so it is an upper bound of the set  $\{\sqcup A_i \mid i \in I\}$ . Since  $\sqcup_{i \in I}(\sqcup A_i)$  is the least upper bound of this set, we have  $\sqcup A \supseteq \sqcup_{i \in I}(\sqcup A_i)$ . Conversely, each  $a \in A$  is contained in some  $A_i$  and therefore below the corresponding  $\sqcup A_i$  which in turn is below  $\sqcup_{i \in I}(\sqcup A_i)$ . Hence the right-hand side is an upper bound of  $A$  and as  $\sqcup A$  is the least such, we also have  $\sqcup A \subseteq \sqcup_{i \in I}(\sqcup A_i)$ . ■

Let us conclude this subsection by looking at an important family of examples of complete lattices. Suppose  $X$  is a set and  $\mathcal{L}$  is a family of subsets of  $X$ . We call  $\mathcal{L}$  a *closure system* if it is closed under the formation of intersections, that is, whenever each member of a family  $(A_i)_{i \in I}$  belongs to  $\mathcal{L}$  then so does  $\bigcap_{i \in I} A_i$ . Because we have allowed the index set to be empty, this implies that  $X$  is in  $\mathcal{L}$ . We call the members of  $\mathcal{L}$  *hulls* or *closed sets*. Given an arbitrary subset  $A$  of  $X$ , one can form  $\bigcap \{B \in \mathcal{L} \mid A \subseteq B\}$ . This is the least superset of  $A$  which belongs to  $\mathcal{L}$  and is called the *hull* or the *closure* of  $A$ .

**Proposition 2.1.5.** *Every closure system is a complete lattice with respect to inclusion.*

**Proof.** Infima are given by intersection and for the supremum one takes the closure of the union. ■

### 2.1.3 Monotone functions

**Definition 2.1.6.** Let  $P$  and  $Q$  be partially ordered sets. A function  $f: P \rightarrow Q$  is called *monotone* if for all  $x, y \in P$  with  $x \subseteq y$  we also have

$f(x) \sqsubseteq f(y)$  in  $Q$ .

‘Monotone’ is really an abbreviation for ‘monotone order-preserving’, but since we have no use for monotone order-reversing maps ( $x \sqsubseteq y \implies f(x) \supseteq f(y)$ ), we have opted for the shorter expression. Alternative terminology is *isotone* (vs. *antitone*) or the other half of the full expression: *order-preserving* mapping.

The set  $[P \xrightarrow{m} Q]$  of all monotone functions between two posets, when ordered *pointwise* (i.e.  $f \sqsubseteq g$  if for all  $x \in P$ ,  $f(x) \sqsubseteq g(x)$ ), gives rise to another partially ordered set, the *monotone function space* between  $P$  and  $Q$ . The category **POSET** of posets and monotone maps has pleasing properties, see Exercise 2.3.9(9).

**Proposition 2.1.7.** *If  $L$  is a complete lattice then every monotone map from  $L$  to  $L$  has a fixpoint. The least of these is given by*

$$\bigcap \{ \hat{x} \in L \mid f(x) \sqsubseteq x \} ,$$

*the largest by*

$$\bigcup \{ x \in L \mid x \sqsubseteq f(x) \} .$$

**Proof.** Let  $A = \{x \in L \mid f(x) \sqsubseteq x\}$  and  $a = \bigcap A$ . For each  $x \in A$  we have  $a \sqsubseteq x$  and  $f(a) \sqsubseteq f(x) \sqsubseteq x$ . Taking the infimum we get  $f(a) \sqsubseteq \bigcap f(A) \sqsubseteq \bigcap A = a$  and  $a \in A$  follows. On the other hand,  $x \in A$  always implies  $f(x) \in A$  by monotonicity. Applying this to  $a$  yields  $f(a) \in A$  and hence  $a \sqsubseteq f(a)$ . ■

For lattices, the converse is also true: The existence of fixpoints for monotone maps implies completeness. But the proof is much harder and relies on the axiom of choice, see [Markowsky, 1976].

#### 2.1.4 Directed sets

**Definition 2.1.8.** Let  $P$  be a poset. A subset  $A$  of  $P$  is *directed*, if it is non-empty and each pair of elements of  $A$  has an upper bound in  $A$ . If a directed set  $A$  has a supremum then this is denoted by  $\bigcup^1 A$ .

Directed lower sets are called *ideals*. Ideals of the form  $\downarrow x$  are called *principal*.

The dual notions are *filtered set* and (*principal*) *filter*.

Simple examples of directed sets are *chains*. These are non-empty subsets which are totally ordered, i.e. for each pair  $x, y$  either  $x \sqsubseteq y$  or  $y \sqsubseteq x$  holds. The chain of natural numbers with their natural order is particularly simple; subsets of a poset isomorphic to it are usually called  $\omega$ -chains. Another frequent type of directed set is given by the set of finite subsets of an arbitrary set. Using this and Proposition 2.1.4(3), we get the following useful decomposition of general suprema.



**Proposition 2.1.9.** *Let  $A$  be a non-empty subset of a  $\sqcup$ -semilattice for which  $\sqcup A$  exists. Then the join of  $A$  can also be written as*

$$\sqcup^{\uparrow} \{ \sqcup M \mid M \subseteq A \text{ finite and non-empty} \}.$$

General directed sets, on the other hand, may be quite messy and unstructured. Sometimes one can find a well-behaved cofinal subset, such as a chain, where we say that  $A$  is *cofinal* in  $B$ , if for all  $b \in B$  there is an  $a \in A$  above it. Such a cofinal subset will have the same supremum (if it exists). But cofinal chains do not always exist, as Exercise 2.3.9(6) shows. Still, every directed set may be thought of as being equipped externally with a nice structure as we will now work out.

**Definition 2.1.10.** A *monotone net* in a poset  $P$  is a monotone function  $\alpha$  from a directed set  $I$  into  $P$ . The set  $I$  is called the *index set* of the net.

Let  $\alpha: I \rightarrow P$  be a monotone net. If we are given a monotone function  $\beta: J \rightarrow I$ , where  $J$  is directed and where for all  $i \in I$  there is  $j \in J$  with  $\beta(j) \geq i$ , then we call  $\alpha \circ \beta: J \rightarrow P$  a *subnet* of  $\alpha$ .

A monotone net  $\alpha: I \rightarrow P$  has a *supremum* in  $P$ , if the set  $\{\alpha(i) \mid i \in I\}$  has a supremum in  $P$ .

Every directed set can be viewed as a monotone net: let the set itself be the index set. On the other hand, the image of a monotone net  $\alpha: I \rightarrow P$  is a directed set in  $P$ . So what are nets good for? The answer is given in the following proposition (which seems to have been stated first in [Krasner, 1939]).

**Lemma 2.1.11.** *Let  $P$  be a poset and let  $\alpha: I \rightarrow P$  be a monotone net. Then  $\alpha$  has a subnet  $\alpha \circ \beta: J \rightarrow P$ , whose index set  $J$  is a lattice in which every principal ideal is finite.*

**Proof.** Let  $J$  be the set of finite subsets of  $I$ . Clearly,  $J$  is a lattice in which every principal ideal is finite. We define the mapping  $\beta: J \rightarrow I$  by induction on the cardinality of the elements of  $J$ :

$$\begin{aligned} \beta(\phi) &= \text{any element of } I; \\ \beta(A) &= \text{any upper bound of the set } A \cup \{\beta(B) \mid B \subset A\}, A \neq \phi. \end{aligned}$$

It is obvious that  $\beta$  is monotone and defines a subnet. ■

This lemma allows to base an induction proof on an arbitrary directed set. This was recently applied to settle a long-standing conjecture in lattice theory, see [Tischendorf and Tůma, 1993].

**Proposition 2.1.12.** *Let  $I$  be directed and  $\alpha: I \times I \rightarrow P$  be a monotone net. Under the assumption that the indicated directed suprema exist, the following equalities hold:*

$$\bigsqcup_{i,j \in I}^{\uparrow} \alpha(i, j) = \bigsqcup_{i \in I}^{\uparrow} (\bigsqcup_{j \in J}^{\uparrow} \alpha(i, j)) = \bigsqcup_{j \in J}^{\uparrow} (\bigsqcup_{i \in I}^{\uparrow} \alpha(i, j)) = \bigsqcup_{i \in I}^{\uparrow} \alpha(i, i).$$

### 2.1.5 Directed-complete partial orders

**Definition 2.1.13.** A poset  $D$  in which every directed subset has a supremum we call a *directed-complete partial order*, or *dcpo* for short.

#### Examples 2.1.14.

- Every complete lattice is also a dcpo. Instances of this are powersets, topologies, subgroup lattices, congruence lattices, and, more generally, closure systems. As Proposition 2.1.9 shows, a lattice which is also a dcpo is almost complete. Only a least element may be missing.
- Every finite poset is a dcpo.
- The set of natural numbers with the usual order does not form a dcpo; we have to add a top element as done in Figure 2. In general, it is a difficult problem how to add points to a poset so that it becomes a dcpo. Using Proposition 2.1.15 below, Markowsky has defined such a completion via chains in [Markowsky, 1976]. Luckily, we need not worry about this problem in domain theory because here we are usually interested in algebraic or continuous dcpos where a completion is easily defined, see Section 2.2.6 below. The correct formulation of what constitutes a completion, of course, also takes morphisms into account. A general framework is described in [Poigné, 1992], Sections 3.3 to 3.6.
- The points of a locale form a dcpo in the specialization order, see [Vickers, 1989; Johnstone, 1982].

More examples will follow in the next subsection. There we will also discuss the question of whether directed sets or  $\omega$ -chains should be used to define dcpo's. Arbitrarily long chains have the full power of directed sets (despite Exercise 2.3.9(6)) as the following proposition shows.

**Proposition 2.1.15.** *A partially ordered set  $D$  is a dcpo if and only if each chain in  $D$  has a supremum.*

The proof, which uses the axiom of choice, goes back to a lemma of Iwamura [1944] and can be found in [Markowsky, 1976].

The following, which may also be found in [Markowsky, 1976], complements Proposition 2.1.7 above.

**Proposition 2.1.16.** *A pointed poset  $P$  is a dcpo if and only if every monotone map on  $P$  has a fixpoint.*

### 2.1.6 Continuous functions

**Definition 2.1.17.** Let  $D$  and  $E$  be dcpos. A function  $f: D \rightarrow E$  is (*Scott-*)*continuous* if it is monotone and if for each directed subset  $A$  of  $D$  we have  $f(\bigsqcup^\uparrow A) = \bigsqcup^\uparrow f(A)$ . We denote the set of all continuous functions from  $D$  to  $E$ , ordered pointwise, by  $[D \longrightarrow E]$ .

A function between pointed dcpos, which preserves the bottom element, is called *strict*. We denote the space of all continuous strict functions by  $[D \xrightarrow{\perp} E]$ .

The identity function on a set  $A$  is denoted by  $\text{id}_A$ , the constant function with image  $\{x\}$  by  $c_x$ .

The preservation of joins of directed sets is actually enough to define continuous maps. In practice, however, one usually needs to show first that  $f(A)$  is directed. This is equivalent to monotonicity.

**Proposition 2.1.18.** Let  $D$  and  $E$  be dcpos. Then  $[D \longrightarrow E]$  is again a dcpo. Directed suprema in  $[D \longrightarrow E]$  are calculated pointwise.

**Proof.** Let  $F$  be a directed collection of functions from  $D$  to  $E$ . Let  $g: D \rightarrow E$  be the function, which is defined by  $g(x) = \bigsqcup_{f \in F}^\uparrow f(x)$ . Let  $A \subseteq D$  be directed.

$$\begin{aligned}
 g(\bigsqcup^\uparrow A) &= \bigsqcup_{f \in F}^\uparrow f(\bigsqcup^\uparrow A) \\
 &= \bigsqcup_{f \in F}^\uparrow \bigsqcup_{a \in A}^\uparrow f(a) \\
 &= \bigsqcup_{a \in A}^\uparrow \bigsqcup_{f \in F}^\uparrow f(a) \\
 &= \bigsqcup_{a \in A}^\uparrow g(a).
 \end{aligned}$$

This shows that  $g$  is continuous. ■

The class of all dcpos together with Scott-continuous functions forms a category, which we denote by **DCPO**. It has strong closure properties as we shall see shortly. For the moment we concentrate on that property of continuous maps which is one of the main reasons for the success of domain theory, namely, that fixpoints can be calculated easily and uniformly.

**Theorem 2.1.19.** Let  $D$  be a pointed dcpo.

1. Every continuous function  $f$  on  $D$  has a least fixpoint. It is given by  $\bigsqcup_{n \in \mathbb{N}} f^n(\perp)$ .
2. The assignment  $\text{fix}: [D \longrightarrow D] \rightarrow D$ ,  $f \mapsto \bigsqcup_{n \in \mathbb{N}}^\uparrow f^n(\perp)$  is continuous.

**Proof.** (1) The set  $\{f^n(\perp) \mid n \in \mathbb{N}\}$  is a chain. This follows from  $\perp \sqsubseteq f(\perp)$  and the monotonicity of  $f$ . Using continuity of  $f$  we get  $f(\bigsqcup_{n \in \mathbb{N}}^{\uparrow} f^n(\perp)) = \bigsqcup_{n \in \mathbb{N}}^{\uparrow} f^{n+1}(\perp)$  and the latter is clearly equal to  $\bigsqcup_{n \in \mathbb{N}}^{\uparrow} f^n(\perp)$ .

If  $x$  is any other fixpoint of  $f$  then from  $\perp \sqsubseteq x$  we get  $f(\perp) \sqsubseteq f(x) = x$  and so on by induction. Hence  $x$  is an upper bound of all  $f^n(\perp)$  and that is why it must be above  $\text{fix}(f)$ .

(2) Let us first look at the  $n$ -fold iteration operator  $\text{it}_n: [D \rightarrow D] \rightarrow D$  which maps  $f$  to  $f^n(\perp)$ . We show its continuity by induction. The 0th iteration operator equals  $c_{\perp}$  so nothing has to be shown there. For the induction step let  $F$  be a directed family of continuous functions on  $D$ . We calculate:

$$\begin{aligned}
 \text{it}_{n+1}(\bigsqcup^{\uparrow} F) &= (\bigsqcup^{\uparrow} F)(\text{it}_n(\bigsqcup^{\uparrow} F)) && \text{definition} \\
 &= (\bigsqcup^{\uparrow} F)(\bigsqcup_{f \in F}^{\uparrow} \text{it}_n(f)) && \text{ind. hypothesis} \\
 &= \bigsqcup_{g \in F}^{\uparrow} g(\bigsqcup_{f \in F}^{\uparrow} (\text{it}_n(f))) && \text{Prop. 2.1.18} \\
 &= \bigsqcup_{g \in F}^{\uparrow} \bigsqcup_{f \in F}^{\uparrow} g(\text{it}_n(f)) && \text{continuity of } g \\
 &= \bigsqcup_{f \in F}^{\uparrow} f^{n+1}(\perp) && \text{Prop. 2.1.12}
 \end{aligned}$$

The pointwise supremum of all iteration operators (which form a chain as we have seen in (1)) is precisely  $\text{fix}$  and so the latter is also continuous. ■

The least fixpoint operator is the mathematical counterpart of recursive and iterative statements in programming languages. When proving a property of such a statement semantically, one often employs the following proof principle which is known under the name *fixpoint induction* (see [Tennent, 1991] or any other book on denotational semantics). Call a predicate on (i.e. a subset of) a dcpo *admissible* if it contains  $\perp$  and is closed under suprema of  $\omega$ -chains. The following is then easily established:

**Lemma 2.1.20.** *Let  $D$  be a dcpo,  $P \subseteq D$  an admissible predicate, and  $f: D \rightarrow D$  a Scott-continuous function. If it is true that  $f(x)$  satisfies  $P$  whenever  $x$  satisfies  $P$ , then it must be true that  $\text{fix}(f)$  satisfies  $P$ .*

We also note the following invariance property of the least fixpoint operator. In fact, it characterizes  $\text{fix}$  uniquely among all fixpoint operators (Exercise 2.3.9(16)).

**Lemma 2.1.21.** *Let  $D$  and  $E$  be pointed dcpos and let*

$$\begin{array}{ccc}
 D & \xrightarrow{h} & E \\
 f \downarrow & & \downarrow g \\
 D & \xrightarrow{h} & E
 \end{array}$$

*be a commutative diagram of continuous functions where  $h$  is strict. Then*



$$\text{fix}(g) = h(\text{fix}(f)).$$

**Proof.** Using continuity of  $h$ , commutativity of the diagram, and strictness of  $h$  in turn, we calculate:

$$\begin{aligned} h(\text{fix}(f)) &= h\left(\bigsqcup_{n \in \mathbb{N}} \uparrow f^n(\perp)\right) \\ &= \bigsqcup_{n \in \mathbb{N}} \uparrow h \circ f^n(\perp) \\ &= \bigsqcup_{n \in \mathbb{N}} \uparrow g^n \circ h(\perp) \\ &= \text{fix}(g) \end{aligned}$$

■

## 2.2 Approximation

In the last subsection we have explained the kind of limits that domain theory deals with, namely, suprema of directed sets. We could have said much more about these ‘convergence spaces’ called dcpos. But the topic can easily become esoteric and lose its connection with computing. For example, the cardinality of dcpos has not been restricted yet and indeed, we didn’t have the tools to sensibly do so (Exercise 2.3.9(18)). We will in this subsection introduce the idea that elements are composed of (or ‘approximated by’) ‘simple’ pieces. This will enrich our theory immensely and will also give the desired connection to semantics.

### 2.2.1 The order of approximation

**Definition 2.2.1.** Let  $x$  and  $y$  be elements of a dcpo  $D$ . We say that  $x$  *approximates*  $y$  if for all directed subsets  $A$  of  $D$ ,  $y \sqsubseteq \bigsqcup \uparrow A$  implies  $x \sqsubseteq a$  for some  $a \in A$ . We say that  $x$  is *compact* if it approximates itself.

We introduce the following notation for  $x, y \in D$  and  $A \subseteq D$ :

$$\begin{aligned} x \ll y &\Leftrightarrow x \text{ approximates } y \\ \downarrow x &= \{y \in D \mid y \ll x\} \\ \uparrow x &= \{y \in D \mid x \ll y\} \\ \uparrow A &= \bigcup_{a \in A} \uparrow a \\ K(D) &= \{x \in D \mid x \text{ compact}\} \end{aligned}$$

The relation  $\ll$  is traditionally called the ‘way-below relation’. M.B. Smyth introduced the expression ‘order of definite refinement’ in [Smyth, 1986]. Throughout this text we will refer to it as the *order of approximation*, even

though the relation is not reflexive. Other common terminology for ‘compact’ is *finite* or *isolated*. The analogy to finite sets is indeed very strong; however one covers a finite set  $M$  by a directed collection  $(A_i)_{i \in I}$  of sets,  $M$  will always be contained in some  $A_i$  already.

In general, approximation is not an absolute property of single points. Rather, we could phrase  $x \ll y$  as ‘ $x$  is a lot simpler than  $y$ ’, which clearly depends on  $y$  as much as it depends on  $x$ .

An element which is compact approximates every element above it. More generally, we observe the following basic properties of approximation.

**Proposition 2.2.2.** *Let  $D$  be a dcpo. Then the following is true for all  $x, x', y, y' \in D$ :*

1.  $x \ll y \implies x \sqsubseteq y$ ;
2.  $x' \sqsubseteq x \ll y \sqsubseteq y' \implies x' \ll y'$ .

### 2.2.2 Bases in dcpos

**Definition 2.2.3.** We say that a subset  $B$  of a dcpo  $D$  is a *basis* for  $D$ , if for every element  $x$  of  $D$  the set  $B_x = \downarrow x \cap B$  contains a directed subset with supremum  $x$ . We call elements of  $B_x$  *approximants to  $x$  relative to  $B$* .

We may think of the rational numbers as a basis for the reals (with a top element added, in order to get a dcpo), but other choices are also possible: dyadic numbers, irrational numbers, etc.

**Proposition 2.2.4.** *Let  $D$  be a dcpo with basis  $B$ .*

1. *For every  $x \in D$  the set  $B_x$  is directed and  $x = \bigsqcup^\uparrow B_x$ .*
2.  *$B$  contains  $K(D)$ .*
3. *Every superset of  $B$  is also a basis for  $D$ .*

**Proof.** (1) It is clear that the join of  $B_x$  equals  $x$ . The point is directedness. From the definition we know there is some directed subset  $A$  of  $B_x$  with  $\bigsqcup^\uparrow A = x$ . Let now  $y, y'$  be elements approximating  $x$ . There must be elements  $a, a'$  in  $A$  above  $y, y'$ , respectively. These have an upper bound  $a''$  in  $A$ , which by definition belongs to  $B_x$ .

(2) We have to show that every element  $c$  of  $K(D)$  belongs to  $B$ . Indeed, since  $c = \bigsqcup^\uparrow B_c$  there must be an element  $b \in B_c$  above  $c$ . All of  $B_c$  is below  $c$ , so  $b$  is actually equal to  $c$ .

(3) is immediate from the definition. ■

**Corollary 2.2.5.** *Let  $D$  be a dcpo with basis  $B$ .*

1. *The largest basis for  $D$  is  $D$  itself.*
2.  *$B$  is the smallest basis for  $D$  if and only if  $B = K(D)$ .*

The ‘only if’ part of (2) is not a direct consequence of the preceding proposition. We leave its proof as Exercise 2.3.9(26).

### 2.2.3 Continuous and algebraic domains

**Definition 2.2.6.** A dcpo is called *continuous* or a *continuous domain* if it has a basis. It is called *algebraic* or an *algebraic domain* if it has a basis of compact elements. We say  $D$  is  $\omega$ -continuous if there exists a countable basis and we call it  $\omega$ -algebraic if  $K(D)$  is a countable basis.

Here we are using the word ‘domain’ for the first time. Indeed, for us a structure only qualifies as a domain if it embodies both a notion of convergence and a notion of approximation.

In the light of Proposition 2.2.4 we can reformulate Definition 2.2.6 as follows, avoiding existential quantification.

**Proposition 2.2.7.**

1. A dcpo  $D$  is continuous if and only if for all  $x \in D$ ,  $x = \bigsqcup^\uparrow \downarrow x$  holds.
2. It is algebraic if and only if for all  $x \in D$ ,  $x = \bigsqcup^\uparrow K(D)_x$  holds.

The word ‘algebraic’ points to algebra. Let us make this connection precise.

**Definition 2.2.8.** A closure system  $\mathcal{L}$  (cf. Section 2.1.2) is called *inductive* if it is closed under directed union.

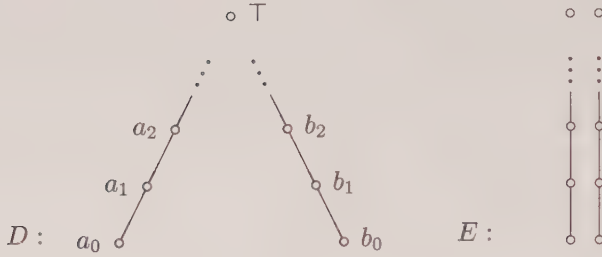
**Proposition 2.2.9.** Every inductive closure system  $\mathcal{L}$  is an algebraic lattice. The compact elements are precisely the finitely generated hulls.

**Proof.** If  $A$  is the hull of a finite set  $M$  and if  $(B_i)_{i \in I}$  is a directed family of hulls such that  $\bigsqcup_{i \in I}^\uparrow B_i = \bigcup_{i \in I} B_i \supseteq A$ , then  $M$  is already contained in some  $B_i$ . Hence hulls of finite sets are compact elements in the complete lattice  $\mathcal{L}$ . On the other hand, every closed set is the directed union of finitely generated hulls, so these form a basis. By Proposition 2.2.4(2), there cannot be any other compact elements. ■

Given a group (or, more generally, an algebra in the sense of universal algebra), then there are two canonical inductive closure systems associated with it, the lattice of subgroups (subalgebras) and the lattice of normal subgroups (congruence relations).

Other standard examples of algebraic domains are:

- Any set with the discrete order is an algebraic domain. In semantics one usually adds a bottom element (standing for divergence) resulting in so-called *flat domains*. (The flat natural numbers are shown in Figure 2.) A basis must in either case contain all elements.
- The set  $[X \multimap Y]$  of partial functions between sets  $X$  and  $Y$  ordered by graph inclusion. Compact elements are those functions which have a finite carrier. It is naturally isomorphic to  $[X \longrightarrow Y_\perp]$  and to  $[X_\perp \xrightarrow{\perp} Y_\perp]$ .
- Every finite poset.



**Fig. 4.** A continuous ( $E$ ) and a non-continuous ( $D$ ) dcpo.

Continuous domains:

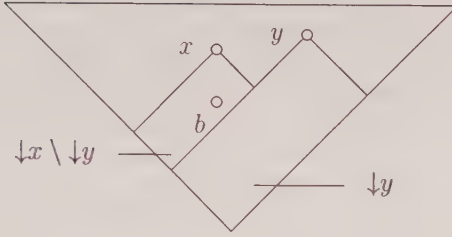
- Every algebraic dcpo is also continuous. This follows directly from the definition. The order of approximation is characterized by  $x \ll y$  if and only if there exists a compact element  $c$  between  $x$  and  $y$ .
- The unit interval is a continuous lattice. It plays a central role in the theory of continuous lattices, see [Gierz *et al.*, 1980], Chapter IV and in particular Theorem 2.19.

Another way of modelling the real numbers in domain theory is to take all closed intervals of finite length and to order them by reversed inclusion. Single element intervals are maximal in this domain and provide a faithful representation of the real line. A countable basis is given by the set of intervals with rational endpoints.

- The lattice of open subsets of a sober space  $X$  forms a continuous lattice if and only if  $X$  is locally compact. Compact Hausdorff spaces are a special case. Here  $O \ll U$  holds if and only if there exists a compact set  $C$  such that  $O \subseteq C \subseteq U$ . This meeting point of topology and domain theory is discussed in detail in [Smyth, 1992; Vickers, 1989; Johnstone, 1982; Gierz *et al.*, 1980] and will also be addressed in Chapter 7.

At this point it may be helpful to give an example of a non-continuous dcpo. The easiest to explain is depicted in Figure 4 (labelled  $D$ ). We show that the order of approximation on  $D$  is empty. Pairs  $(a_i, b_j)$  and  $(b_i, a_j)$  cannot belong to the order of approximation because they are not related in the order. Two points  $a_i \sqsubseteq a_j$  in the same ‘leg’ are still not approximating because  $(b_n)_{n \in \mathbb{N}}$  is a directed set with supremum above  $a_j$  but containing no element above  $a_i$ .

A non-continuous distributive complete lattice is much harder to visualize by a line diagram. From what we have said we know that the topology of a sober space which is not locally compact is such a lattice. Exercise 2.3.9(21) discusses this in detail.



**Fig. 5.** Basis element  $b$  witnesses that  $x$  is not below  $y$ .

If  $D$  is pointed then the order of approximation is non-empty because a bottom element approximates every other element.

A basis not only gives approximations for elements, it also approximates the order relation:

**Proposition 2.2.10.** *Let  $D$  be a continuous domain with basis  $B$  and let  $x$  and  $y$  be elements of  $D$ . Then  $x \sqsubseteq y$ ,  $B_x \subseteq B_y$  and  $B_x \subseteq \downarrow y$  are all equivalent.*

The form in which we will usually apply this proposition is:  $x \not\sqsubseteq y$  implies there exists  $b \in B_x$  with  $b \not\sqsubseteq y$ . A picture of this situation is given in Figure 5.

In the light of Proposition 2.2.10 we can now also give a more intuitive reason why the dcpo  $D$  in Figure 4 is not continuous. A natural candidate for a basis in  $D$  is the collection of all  $a_i$ s and  $b_i$ s (certainly,  $\top$  doesn't approximate anything). Proposition 2.2.10 expresses the idea that in a continuous domain all information about how elements are related is contained in the basis already. And the fact that  $\bigsqcup_{n \in \mathbb{N}} a_n = \bigsqcup_{n \in \mathbb{N}} b_n = \top$  holds in  $D$  is precisely what is not visible in the would-be basis. Thus, the dcpo should look rather like  $E$  in the same figure (which indeed is an algebraic domain).

Bases allow one to express the continuity of functions in a form reminiscent of the  $\epsilon$ - $\delta$  definition for real-valued functions.

**Proposition 2.2.11.** *A map  $f$  between continuous domains  $D$  and  $E$  with bases  $B$  and  $C$ , respectively, is continuous if and only if for each  $x \in D$  and  $e \in C_{f(x)}$  there exists  $d \in B_x$  with  $f(\uparrow d) \subseteq \uparrow e$ .*

**Proof.** By continuity we have  $f(x) = f(\bigsqcup^\uparrow B_x) = \bigsqcup_{d \in B_x}^\uparrow f(d)$ . Since  $e$  approximates  $f(x)$ , there exists  $d \in B_x$  with  $f(d) \sqsupseteq e$ . Monotonicity of  $f$  then implies  $f(\uparrow d) \subseteq \uparrow e$ .

For the converse we first show monotonicity. Suppose  $x \sqsubseteq y$  holds but  $f(x)$  is not below  $f(y)$ . By Proposition 2.2.10 there is  $e \in C_{f(x)} \setminus \downarrow f(y)$  and



from our assumption we get  $d \in B_x$  such that  $f(\uparrow d) \subseteq \uparrow e$ . Since  $y$  belongs to  $\uparrow d$  this is a contradiction. Now let  $A$  be a directed subset of  $D$  with  $x$  as its join. Monotonicity implies  $\bigsqcup^\uparrow f(A) \subseteq f(\bigsqcup^\uparrow A) = f(x)$ . If the converse relation does not hold then we can again choose  $e \in C_{f(x)}$  with  $e \not\subseteq \bigsqcup^\uparrow f(A)$  and for some  $d \in B_x$  we have  $f(\uparrow d) \subseteq \uparrow e$ . Since  $d$  approximates  $x$ , some  $a \in A$  is above  $d$  and we get  $\bigsqcup^\uparrow f(A) \supseteq f(a) \supseteq f(d) \supseteq e$ , contradicting our choice of  $e$ . ■

Finally, we cite a result which reduces the calculation of least fixpoints to a basis. The point here is that a continuous function need not preserve compactness nor the order of approximation and so the sequence  $\perp, f(\perp), f(f(\perp)), \dots$  need not consist of basis elements.

**Proposition 2.2.12.** *If  $D$  is a pointed  $\omega$ -continuous domain with basis  $B$  and if  $f: D \rightarrow D$  is a continuous map, then there exists an  $\omega$ -chain  $b_0 \subseteq b_1 \subseteq b_2 \subseteq \dots$  of basis elements such that the following conditions are satisfied:*

1.  $b_0 = \perp$ ,
2.  $\forall n \in \mathbb{N}. b_{n+1} \subseteq f(b_n)$ ,
3.  $\bigsqcup_{n \in \mathbb{N}}^\uparrow b_n = \text{fix}(f) \quad (= \bigsqcup_{n \in \mathbb{N}}^\uparrow f^n(\perp))$ .

A proof may be found in [Abramsky, 1990b].

## 2.2.4 Comments on possible variations

**Directed sets vs.  $\omega$ -chains** Let us start with the following observation.

**Proposition 2.2.13.** *If a dcpo  $D$  has a countable basis then every directed subset of  $D$  contains an  $\omega$ -chain with the same supremum.*

This raises the question whether one shouldn't build up the whole theory using  $\omega$ -chains. The basic definitions then read: An  $\omega$ -ccpo is a poset in which every  $\omega$ -chain has a supremum. A function is  $\omega$ -continuous if it preserves joins of  $\omega$ -chains. An element  $x$  is  $\omega$ -approximating  $y$  if  $\bigsqcup_{n \in \mathbb{N}}^\uparrow a_n \supseteq y$  implies  $a_n \supseteq x$  for some  $n \in \mathbb{N}$ . An  $\omega$ -ccpo is continuous if there is a countable subset  $B$  such that every element is the join of an  $\omega$ -chain of elements from  $B$   $\omega$ -approximating it. Similarly for algebraicity. (This is the approach adopted in [Plotkin, 1981], for example.) The main point about these definitions is the countability of the basis. It ensures that they are in complete harmony with our set-up, because we can show:

**Proposition 2.2.14.**

1. *Every continuous  $\omega$ -ccpo is a continuous dcpo.*
2. *Every algebraic  $\omega$ -ccpo is an algebraic dcpo.*
3. *Every  $\omega$ -continuous map between continuous  $\omega$ -ccpos is continuous.*

**Proof.** (1) Let  $(b_n)_{n \in \mathbb{N}}$  be an enumeration of a basis  $B$  for  $D$ . We first show that the continuous  $\omega$ -ccpo  $D$  is directed-complete, so let  $A$  be a

directed subset of  $D$ . Let  $B'$  be the set of basis elements which are below some element of  $A$  and, for simplicity, assume that  $B = B'$ . We construct an  $\omega$ -chain in  $A$  as follows: let  $a_0$  be an element of  $A$  which is above  $b_0$ . Then let  $b_{n_1}$  be the first basis element not below  $a_0$ . It must be below some  $a'_1 \in A$  and we set  $a_1$  to be an upper bound of  $a_0$  and  $a'_1$  in  $A$ . We proceed by induction. It does not follow that the resulting chain  $(a_n)_{n \in \mathbb{N}}$  is cofinal in  $A$  but it is true that its supremum is also the supremum of  $A$ , because both subsets of  $D$  dominate the same set of basis elements.

This construction also shows that  $\omega$ -approximation is the same as approximation in a continuous  $\omega$ -ccpo. The same basis  $B$  may then be used to show that  $D$  is a continuous domain. (The directedness of the sets  $B_x$  follows as in Proposition 2.2.4(1).)

(2) follows from the proof of (1), so it remains to show (3). Monotonicity of the function  $f$  is implied in the definition of  $\omega$ -continuity. Therefore a directed set  $A \subseteq D$  is mapped onto a directed set in  $E$  and also  $f(\bigsqcup^\uparrow A) \sqsubseteq \bigsqcup^\uparrow f(A)$  holds. Let  $(a_n)_{n \in \mathbb{N}}$  be an  $\omega$ -chain in  $A$  with  $\bigsqcup^\uparrow A = \bigsqcup^\uparrow_{n \in \mathbb{N}} a_n$ , as constructed in the proof of (1). Then we have  $f(\bigsqcup^\uparrow A) = f(\bigsqcup^\uparrow_{n \in \mathbb{N}} a_n) = \bigsqcup^\uparrow_{n \in \mathbb{N}} f(a_n) \sqsubseteq \bigsqcup^\uparrow f(A)$ . ■

If we drop the crucial assumption about the countability of the basis then the two theories bifurcate and, in our opinion, the theory based on  $\omega$ -chains becomes rather bizarre. To give just one illustration, observe that simple objects, such as powersets, may fail to be algebraic domains. There remains the question, however, whether in the realm of a mathematical theory of computation one should start with  $\omega$ -chains. Arguments in favor of this approach point to pedagogy and foundations. The pedagogical aspect is somewhat weakened by the fact that even in a continuous  $\omega$ -ccpo the sets  $\downarrow x$  happen to be directed. Glossing over this fact would tend to mislead the student. In our eyes, the right middle ground for a *course* on domain theory, then, would be to start with  $\omega$ -chains and motivations from semantics and then at some point (probably where the ideal completion of a poset is discussed) to switch to directed sets as the more general concept. This suggestion is hardly original. It is in direct analogy with the way students are introduced to topological concepts.

Turning to foundations, we feel that the necessity to *choose* chains where directed subsets are naturally available (such as in function spaces) and thus to rely on the axiom of choice without need, is a serious stain on this approach. To take foundational questions seriously implies a much deeper re-working of the theory: some pointers to the literature will be found in Section 8.

We do not feel the need to say much about the use of chains of arbitrary cardinality. This adds nothing in strength (because of Proposition 2.1.15) but has all the disadvantages pointed out for  $\omega$ -chains already.

**Bases vs. intrinsic descriptions.** The definition of a continuous do-

main given here differs from, and is in fact more complicated than the standard one (which we presented as Proposition 2.2.7(1)). We nevertheless preferred this approach to the concept of approximation for three reasons. Firstly, the standard definition does not allow the restriction of the size of continuous domains. In this respect not the cardinality of a domain but the minimal cardinality of a basis is of interest. Secondly, we wanted to point out the strong analogy between algebraic and continuous domains. And, indeed, the proofs we have given so far for continuous domains specialize directly to the algebraic case if one replaces ' $B$ ' by ' $K(D)$ ' throughout. Thus far at least, proofs for algebraic domains alone would not be any shorter. And, thirdly, we wanted to stress the idea of approximation by elements which are (for whatever reason) simpler than others. Such a notion of simplicity does often exist for continuous domains (such as rational vs. real numbers), even though its justification is not purely order-theoretical (see 8.1.1).

**Algebraic vs. continuous.** This brings up the question of why one bothers with continuous domains at all. There are two important reasons but they depend on definitions introduced later in this text. The first is the simplification of the mathematical theory of domains stemming from the possibility of freely using retracts (see Theorem 3.1.4 below). The second is the observation that in algebraic domains two fundamental concepts of domain theory essentially coincide, namely, that of a Scott-open set and that of a compact saturated set. We find it pedagogically advantageous to be able to distinguish between the two.

**Continuous dcpo vs. continuous domain.** It is presently common practice to start a paper in semantics or domain theory by defining the subclass of dcpo's of interest and then assigning the name 'domain' to these structures. We fully agree with this custom of using 'domain' as a generic name. In this article, however, we will study a full range of possible definitions, the most general of which is that of a dcpo. We have nevertheless avoided calling these domains. For us, 'domain' refers to both ideas essential to the theory, namely, the idea of convergence and the idea of approximation.

### 2.2.5 Useful properties

Let us start right away with the single most important feature of the order of approximation, the *interpolation property*.

**Lemma 2.2.15.** *Let  $D$  be a continuous domain and let  $M \subseteq D$  be a finite set each of whose elements approximates  $y$ . Then there exists  $y' \in D$  such that  $M \ll y' \ll y$  holds. If  $B$  is a basis for  $D$ , then  $y'$  may be chosen from  $B$ . (We say  $y'$  interpolates between  $M$  and  $y$ .)*

**Proof.** Given  $M \ll y$  in  $D$  we define the set

$$A = \{a \in D \mid \exists a' \in D : a \ll a' \ll y\}.$$

It is clearly non-empty. It is directed because if  $a \ll a' \ll y$  and  $b \ll b' \ll y$ , then by the directedness of  $\downarrow y$  there is  $c' \in D$  such that  $a' \sqsubseteq c' \ll y$  and  $b' \sqsubseteq c' \ll y$  and again by the directedness of  $\downarrow c'$  there is  $c \in D$  with  $a \sqsubseteq c \ll c'$  and  $b \sqsubseteq c \ll c'$ . We calculate the supremum of  $A$ : let  $y'$  be any element approximating  $y$ . Since  $\downarrow y' \subseteq A$  we have that  $\bigsqcup^\uparrow A \supseteq \bigsqcup^\uparrow \downarrow y' = y'$ . This holds for all  $y' \ll y$  so by continuity  $y = \bigsqcup^\uparrow \downarrow y \subseteq \bigsqcup^\uparrow A$ . All elements of  $A$  are less than  $y$ , so in fact equality holds:  $\bigsqcup^\uparrow \downarrow y = \bigsqcup^\uparrow A$ . Remember that we started out with a set  $M$  whose elements approximate  $y$ . By definition there is  $a_m \in A$  with  $m \sqsubseteq a_m$  for each  $m \in M$ . Let  $a$  be an upper bound of the  $a_m$  in  $A$ . By definition, for some  $a'$ ,  $a \ll a' \ll y$ , and we can take  $a'$  as an interpolating element between  $M$  and  $y$ . The proof remains the same if we allow only basis elements to enter  $A$ . ■

**Corollary 2.2.16.** *Let  $D$  be a continuous domain with a basis  $B$  and let  $A$  be a directed subset of  $D$ . If  $c$  is an element approximating  $\bigsqcup^\uparrow A$ , then  $c$  already approximates some  $a \in A$ . As a formula:*

$$\downarrow \bigsqcup^\uparrow A = \bigcup_{a \in A} \downarrow a.$$

*Intersecting with the basis on both sides gives*

$$B_{\bigsqcup^\uparrow A} = \bigcup_{a \in A} B_a.$$

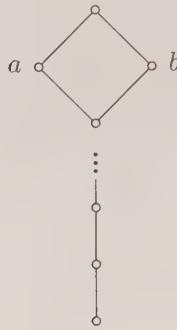
Next we will illustrate how in a domain we can restrict attention to principal ideals.

**Proposition 2.2.17.**

1. *If  $D$  is a continuous domain and if  $x, y$  are elements in  $D$ , then  $x$  approximates  $y$  if and only if for all directed sets  $A$  with  $\bigsqcup^\uparrow A = y$  there is an  $a \in A$  such that  $a \supseteq x$ .*
2. *The order of approximation on a continuous domain is the union of the orders of approximation on all principal ideals.*
3. *A dcpo is continuous if and only if each principal ideal is continuous.*
4. *For a continuous domain  $D$  we have  $K(D) = \bigcup_{x \in D} K(\downarrow x)$ .*
5. *A dcpo is algebraic if and only if each principal ideal is algebraic.*

**Proposition 2.2.18.**

1. *In a continuous domain minimal upper bounds of finite sets of compact elements are again compact.*
2. *In a complete lattice the sets  $\downarrow x$  are  $\sqcup$ -sub-semilattices.*



**Fig. 6.** The meet of the compact elements  $a$  and  $b$  is not compact.

3. In a complete lattice the join of finitely many compact elements is again compact.

**Corollary 2.2.19.** *A complete lattice is algebraic if and only if each element is the join of compact elements.*

The infimum of compact elements need not be compact again, even in an algebraic lattice. An example is given in Figure 6.

### 2.2.6 Bases as objects

In Section 2.2.2 we have seen how we can use bases in order to express properties of the ambient domain. We will now study the question of how far we can reduce domain theory to a theory of (abstract) bases. The resulting techniques will prove useful in later chapters but we hope that they will also deepen the reader's understanding of the nature of domains.

We start with the question of what additional information is necessary in order to reconstruct a domain from one of its bases. Somewhat surprisingly, it is just the order of approximation. Thus we define:

**Definition 2.2.20.** An (abstract) *basis* is given by a set  $B$  together with a transitive relation  $\prec$  on  $B$ , such that

$$(INT) \quad M \prec x \implies \exists y \in B. M \prec y \prec x$$

holds for all elements  $x$  and finite subsets  $M$  of  $B$ .

Abstract bases were introduced in [Smyth, 1977] where they are called 'R-structures'. Examples of abstract bases are concrete bases of continuous domains, of course, where the relation  $\prec$  is the restriction of the order of approximation. Axiom (INT) is satisfied because of Lemma 2.2.15 and because we have required bases in domains to have directed sets of approximants for each element.



Other examples are partially ordered sets, where (INT) is satisfied because of reflexivity. We will shortly identify posets as being exactly the bases of compact elements of algebraic domains.

In what follows we will use the terminology developed at the beginning of this chapter, even though the relation  $\prec$  on an abstract basis need neither be reflexive nor antisymmetric. This is convenient but in some instances looks more innocent than it is. An ideal  $A$  in a basis, for example, has the property (following from directedness) that for every  $x \in A$  there is another element  $y \in A$  with  $x \prec y$ . In posets this doesn't mean anything but here it becomes an important feature. Sometimes this is stressed by using the expression ' $A$  is a *round* ideal'. Note that a set of the form  $\downarrow x$  is always an ideal because of (INT) but that it need not contain  $x$  itself. We will refrain from calling  $\downarrow x$  'principal' in these circumstances.

**Definition 2.2.21.** For a basis  $\langle B, \prec \rangle$  let  $\text{Idl}(B)$  be the set of all ideals ordered by inclusion. It is called the *ideal completion* of  $B$ . Furthermore, let  $i: B \rightarrow \text{Idl}(B)$  denote the function which maps  $x \in B$  to  $\downarrow x$ . If we want to stress the relation with which  $B$  is equipped then we write  $\text{Idl}(B, \prec)$  for the ideal completion.

**Proposition 2.2.22.** Let  $\langle B, \prec \rangle$  be an abstract basis.

1. The ideal completion of  $B$  is a dcpo.
2.  $A \ll A'$  holds in  $\text{Idl}(B)$  if and only if there are  $x \prec y$  in  $B$  such that  $A \subseteq i(x) \subseteq i(y) \subseteq A'$ .
3.  $\text{Idl}(B)$  is a continuous domain and a basis of  $\text{Idl}(B)$  is given by  $i(B)$ .
4. If  $\prec$  is reflexive then  $\text{Idl}(B)$  is algebraic.
5. If  $\langle B, \prec \rangle$  is a poset then  $B$ ,  $K(\text{Idl}(B))$ , and  $i(B)$  are all isomorphic.

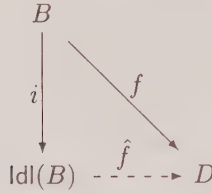
**Proof.** (1) holds because clearly the directed union of ideals is an ideal. Roundness implies that every  $A \in \text{Idl}(B)$  can be written as  $\bigcup_{x \in A} \downarrow x$ . This union is directed because  $A$  is directed. This proves (2) and also (3). The fourth claim follows from the characterization of the order of approximation. The last clause holds because there is only one basis of compact elements for an algebraic domain. ■

Defining the product of two abstract bases as one does for partially ordered sets, we have the following:

**Proposition 2.2.23.**  $\text{Idl}(B \times B') \cong \text{Idl}(B) \times \text{Idl}(B')$ .

Our 'completion' has a weak universal property:

**Proposition 2.2.24.** Let  $\langle B, \prec \rangle$  be an abstract basis and let  $D$  be a dcpo. For every monotone function  $f: B \rightarrow D$  there is a largest continuous function  $\hat{f}: \text{Idl}(B) \rightarrow D$  such that  $\hat{f} \circ i$  is below  $f$ . It is given by  $\hat{f}(A) = \bigsqcup^1 f(A)$ .



The assignment  $f \mapsto \hat{f}$  is a Scott-continuous map from  $[B \xrightarrow{m} D]$  to  $[\text{Idl}(B) \rightarrow D]$ .

If the relation  $\prec$  is reflexive then  $\hat{f} \circ i$  equals  $f$ .

**Proof.** Let us first check continuity of  $\hat{f}$ . To this end let  $(A_i)_{i \in I}$  be a directed collection of ideals. Using general associativity (Proposition 2.1.4(3)) we can calculate:  $\hat{f}(\bigsqcup_{i \in I}^\uparrow A_i) = \hat{f}(\bigcup_{i \in I} A_i) = \bigsqcup^\uparrow \{f(x) \mid x \in \bigcup_{i \in I} A_i\} = \bigsqcup_{i \in I}^\uparrow \bigsqcup^\uparrow \{f(x) \mid x \in A_i\} = \bigsqcup_{i \in I}^\uparrow \hat{f}(A_i)$ .

Since  $f$  is assumed to be monotone,  $f(x)$  is an upper bound for  $f(\downarrow x)$ . This proves that  $\hat{f} \circ i$  is below  $f$ . If, on the other hand,  $g: \text{Idl}(B) \rightarrow D$  is another continuous function with this property then we have  $g(A) = g(\bigcup_{x \in A} \downarrow x) = \bigsqcup_{x \in A}^\uparrow g(\downarrow x) = \bigsqcup_{x \in A}^\uparrow g(i(x)) \subseteq \bigsqcup_{x \in A}^\uparrow f(x) = \hat{f}(A)$ .

The claim about the continuity of the assignment  $f \mapsto \hat{f}$  is shown by the usual switch of directed suprema.

If  $\prec$  is a preorder then we can show that  $\hat{f} \circ i = f$ :  $\hat{f}(i(x)) = \hat{f}(\downarrow x) = \bigsqcup^\uparrow f(\downarrow x) = f(x)$ . ■

A particular instance of this proposition is the case that  $B$  and  $B'$  are two abstract bases and  $f: B \rightarrow B'$  is monotone. By the extension of  $f$  to  $\text{Idl}(B)$  we mean the map  $\widehat{i' \circ f}: \text{Idl}(B) \rightarrow \text{Idl}(B')$ . It maps an ideal  $A \subseteq B$  to the ideal  $\downarrow f(A)$ .

**Proposition 2.2.25.** *Let  $D$  be a continuous domain with basis  $B$ . Viewing  $\langle B, \ll \rangle$  as an abstract basis, we have the following:*

1.  $\text{Idl}(B)$  is isomorphic to  $D$ . The isomorphism  $\sigma: \text{Idl}(B) \rightarrow D$  is the extension  $\hat{e}$  of the embedding of  $B$  into  $D$ . Its inverse  $\beta$  maps elements  $x \in D$  to  $B_x$ .
2. For every dcpo  $E$  and continuous function  $f: D \rightarrow E$  we have  $f = \hat{g} \circ \beta$ , where  $g$  is the restriction of  $f$  to  $B$ .

**Proof.** In a continuous domain we have  $x = \bigsqcup^\uparrow B_x$  for all elements, so  $\sigma \circ \beta = \text{id}_D$ . Composing the maps the other way round we need to see that every  $c \in B$  which approximates  $\bigsqcup^\uparrow A$ , where  $A$  is an ideal in  $\langle B, \ll \rangle$ , actually belongs to  $A$ . We interpolate:  $c \ll d \ll \bigsqcup^\uparrow A$  and using the defining property of the order of approximation, we find  $a \in A$  above  $d$ . Therefore  $c$  approximates  $a$  and belongs to  $A$ .

The calculation for (2) is straightforward:

$$f(x) = f(\bigsqcup^\uparrow B_x) = \bigsqcup^\uparrow f(B_x) = \hat{g}(B_x) = \hat{g}(\beta(x)).$$

■

**Corollary 2.2.26.** *A continuous function from a continuous domain  $D$  to a dcpo  $E$  is completely determined by its behavior on a basis of  $D$ .*

As we now know how to reconstruct a continuous domain from its basis and how to recover a continuous function from its restriction to the basis, we may wonder whether it is possible to work with bases alone. There is one further problem to overcome, namely the fact that continuous functions do not preserve the order of approximation. The only way out is to switch from functions to relations, where we relate a basis element  $c$  to all basis elements approximating  $f(c)$ . This can be axiomatized as follows.

**Definition 2.2.27.** A relation  $R$  between abstract bases  $B$  and  $C$  is called *approximable* if the following conditions are satisfied:

1.  $\forall x \in B \forall y, y' \in C. (xRy \succ y' \implies xRy');$
2.  $\forall x \in B \forall M \subseteq_{\text{fin}} C. (\forall y \in M. xRy \implies (\exists z \in C. xRz \text{ and } z \succ M));$
3.  $\forall x, x' \in B \forall y \in C. (x' \succ xRy \implies x'Ry);$
4.  $\forall x \in B \forall y \in C. (xRy \implies (\exists z \in B. x \succ zRy)).$

The following is then proved without difficulty.

**Theorem 2.2.28.** *The category of abstract bases and approximable relations is equivalent to **CONT**, the category of continuous dcpos and continuous maps.*

The formulations we have chosen in this section allow us to immediately read off the corresponding results in the special case of algebraic domains. In particular:

**Theorem 2.2.29.** *The category of pre-orders and approximable relations is equivalent to **ALG**, the category of algebraic dcpos and continuous maps.*

## 2.3 Topology

By a *topology* on a space  $X$  we understand a system of subsets of  $X$  (called the *open sets*), which is closed under finite intersections and infinite unions. It is an amazing fact that by a suitable choice of a topology we can encode all information about convergence, approximation, continuity of functions, and even points of  $X$  themselves. To a student of mathematics this appears to be an immense abstraction from the intuitive beginnings of analysis. In domain theory we are in the lucky situation that we can tie up open sets with the concrete idea of observable properties. This was done in detail in Vol. 1 of this Handbook, [Smyth, 1992], and we may therefore proceed swiftly to the mathematical side of the subject.

### 2.3.1 The Scott-topology on a dcpo

**Definition 2.3.1.** Let  $D$  be a dcpo. A subset  $A$  is called (*Scott-*)closed if it is a lower set and is closed under suprema of directed subsets. Complements of closed sets are called (*Scott-*)open; they are the elements of  $\sigma_D$ , the *Scott-topology* on  $D$ .

We shall use the notation  $\text{Cl}(A)$  for the smallest closed set containing  $A$ . Similarly,  $\text{Int}(A)$  will stand for the open kernel of  $A$ .

A Scott-open set  $O$  is necessarily an upper set. By contraposition it is characterized by the property that every directed set whose supremum lies in  $O$  has a non-empty intersection with  $O$ .

Basic examples of closed sets are principal ideals. This knowledge is enough to show the following:

**Proposition 2.3.2.** *Let  $D$  be a dcpo.*

1. *For elements  $x, y \in D$  the following are equivalent:*
  - (a)  $x \sqsubseteq y$ ,
  - (b) *every Scott-open set which contains  $x$  also contains  $y$ ,*
  - (c)  $x \in \text{Cl}(\{y\})$ .
2. *The Scott-topology satisfies the  $T_0$  separation axiom.*
3.  *$\langle D, \sigma_D \rangle$  is a Hausdorff ( $= T_2$ ) topological space if and only if the order on  $D$  is trivial.*

Thus we can reconstruct the order between elements of a dcpo from the Scott-topology. The same is true for limits of directed sets.

**Proposition 2.3.3.** *Let  $A$  be a directed set in a dcpo  $D$ . Then  $x \in D$  is the supremum of  $A$  if and only if it is an upper bound for  $A$  and every Scott-neighbourhood of  $x$  contains an element of  $A$ .*

**Proof.** Indeed, the closed set  $\downarrow \bigcup^\uparrow A$  separates the supremum from all other upper bounds of  $A$ . ■

**Proposition 2.3.4.** *For dcpos  $D$  and  $E$ , a function  $f$  from  $D$  to  $E$  is Scott-continuous if and only if it is topologically continuous with respect to the Scott-topologies on  $D$  and  $E$ .*

**Proof.** Let  $f$  be a continuous function from  $D$  to  $E$  and let  $O$  be an open subset of  $E$ . It is clear that  $f^{-1}(O)$  is an upper set because continuous functions are monotone. If  $f$  maps the element  $x = \bigcup^\uparrow_{i \in I} x_i \in D$  into  $O$  then we have  $f(x) = f(\bigcup^\uparrow_{i \in I} x_i) = \bigcup^\uparrow_{i \in I} f(x_i) \in O$  and by definition there must be some  $x_i$  which is mapped into  $O$ . Hence  $f^{-1}(O)$  is open in  $D$ .

For the converse assume that  $f$  is topologically continuous. We first show that  $f$  must be monotone: Let  $x \sqsubseteq x'$  be elements of  $D$ . The inverse image of the Scott-closed set  $\downarrow f(x')$  contains  $x'$ . Hence it also contains  $x$ . Now let  $A \subseteq D$  be directed. Look at the inverse image of the Scott-closed set  $\downarrow (\bigcup^\uparrow_{a \in A} f(a))$ . It contains  $A$  and is Scott-closed, too. So it must also

contain  $\bigsqcup^\uparrow A$ . Since by monotonicity  $f(\bigsqcup^\uparrow A)$  is an upper bound of  $f(A)$ , it follows that  $f(\bigsqcup^\uparrow A)$  is the supremum of  $f(A)$ . ■

So much for the theme of convergence. Let us now proceed to see how far approximation is reflected in the Scott-topology.

### 2.3.2 The Scott-topology on domains

In this subsection we work with the second most primitive form of open sets, namely those which can be written as  $\uparrow x$ . We start by characterizing the order of approximation.

**Proposition 2.3.5.** *Let  $D$  be a continuous domain. Then the following are equivalent for all pairs  $x, y \in D$ :*

1.  $x \ll y$ ,
2.  $y \in \text{Int}(\uparrow x)$ ,
3.  $y \in \uparrow x$ .

**Proposition 2.3.6.** *Let  $D$  be a continuous domain with basis  $B$ . Then openness of a subset  $O$  of  $D$  can be characterized in the following two ways:*

1.  $O = \bigcup_{x \in O} \uparrow x$ ,
2.  $O = \bigcup_{x \in O \cap B} \uparrow x$ .

This can be read as saying that every open set is supported by its members from the basis. We may therefore ask how the Scott-topology is derived from an abstract basis.

**Proposition 2.3.7.** *Let  $(B, \prec)$  be an abstract basis and let  $M$  be any subset of  $B$ . Then the set  $\{A \in \text{Idl}(B) \mid M \cap A \neq \emptyset\}$  is Scott-open in  $\text{Idl}(B)$  and all open sets on  $\text{Idl}(B)$  are of this form.*

This, finally, nicely connects the theory up with the idea of an observable property. If we assume that the elements of an abstract basis are finitely describable and finitely recognizable (and we strongly approve of this intuition) then it is clear how to observe a property in the completion: we have to wait until we see an element from a given set of basis elements.

We also have the following sharpening of Proposition 2.3.6:

**Lemma 2.3.8.** *Every Scott-open set in a continuous domain is a union of Scott-open filters.*

**Proof.** Let  $x$  be an element in the open set  $O$ . By Proposition 2.3.6 there is an element  $y \in O$  which approximates  $x$ . We repeatedly interpolate between  $y$  and  $x$ . This gives us a sequence  $y \ll \dots \ll y_n \ll \dots \ll y_1 \ll x$ . The union of all  $\uparrow y_n$  is a Scott-open filter containing  $x$  and contained in  $O$ . ■

In this subsection we have laid the groundwork for a formulation of domain theory purely in terms of the lattice of Scott-open sets. Since we



construe open sets as properties we have also brought logic into the picture. This relationship will be looked at more closely in Chapter 7. There and in Section 4.2.3 we will also exhibit more properties of the Scott-topology on domains.

### Exercises 2.3.9.

1. Formalize the passage from pre-orders to their quotient posets.
2. Draw line diagrams of the powersets of a one, two, three, and four element set.
3. Show that a poset which has all suprema also has all infima, and vice versa.
4. Refine Proposition 2.1.7 by showing that the fixpoints of a monotone function on a complete lattice form a complete lattice. Is it a sublattice?
5. Show that finite directed sets have a largest element. Characterize the class of posets in which this is true for every directed set.
6. Show that the directed set of finite subsets of real numbers does not contain a cofinal chain.
7. Which of the following are dcpos:  $\mathbb{R}$ ,  $[0, 1]$  (unit interval),  $\mathbb{Q}$ ,  $\mathbb{Z}^-$  (negative integers)?
8. Let  $f$  be a monotone map between complete lattices  $L$  and  $M$  and let  $A$  be a subset of  $L$ . Prove:  $f(\bigsqcup A) \supseteq \bigsqcup f(A)$ .
9. Show that the category of posets and monotone functions forms a cartesian closed category.
10. Draw the line diagram for the function space of the flat booleans (see Figure 1).
11. Show that an ideal in a (binary) product of posets can always be seen as the product of two ideals from the individual posets.
12. Show that a map  $f$  between two dcpos  $D$  and  $E$  is continuous if and only if for all directed sets  $A$  in  $D$ ,  $f(\bigsqcup^\uparrow A) = \bigsqcup f(A)$  holds.
13. Give an example of a monotone map  $f$  on a pointed dcpo  $D$  for which  $\bigsqcup_{n \in \mathbb{N}}^\uparrow f^n(\perp)$  is not a fixpoint. (Some fixpoint must exist by Proposition 2.1.16.)
14. Use fixpoint induction to prove the following. Let  $f, g: D \rightarrow D$  be continuous functions on a pointed dcpo  $D$  with  $f(\perp) = g(\perp)$ , and  $f \circ g = g \circ f$ . Then  $\text{fix}(f) = \text{fix}(g)$ .
15. (Dinaturality of fixpoints) Let  $D, E$  be pointed dcpos and let  $f: D \rightarrow E, g: E \rightarrow D$  be continuous functions. Prove

$$\text{fix}(g \circ f) = g(\text{fix}(f \circ g)) .$$

16. Show that Lemma 2.1.21 uniquely characterizes  $\text{fix}$  among all fixpoint operators.

17. Prove: Given pointed dcpo's  $D$  and  $E$  and a continuous function  $f: D \times E \rightarrow E$  there is a continuous function  $Y(f): D \rightarrow E$  such that  $Y(f) = f \circ \langle \text{id}_D, Y(f) \rangle$  holds. (This is the general definition of a category having fixpoints.) How does Theorem 2.1.19 follow from this?
18. Show that each version of the natural numbers as shown in Figure 2 is an example of a countable dcpo whose function space is uncountable.
19. Characterize the order of approximation on the unit interval. What are the compact elements?
20. Show that in finite posets every element is compact.
21. Let  $L$  be the lattice of open sets of  $\mathbb{Q}$ , where  $\mathbb{Q}$  is equipped with the ordinary metric topology. Show that no two non-empty open sets approximate each other. Conclude that  $L$  is not continuous.
22. Prove Proposition 2.2.10.
23. Extend Proposition 2.2.10 in the following way: For every finite subset  $M$  of a continuous dcpo  $D$  with basis  $B$  there exists  $M' \subseteq B$ , such that  $x \mapsto x'$  is an order-isomorphism between  $M$  and  $M'$  and such that for all  $x \in M$ , the element  $x'$  belongs to  $B_x$ .
24. Prove Proposition 2.2.17.
25. Show that elements of an abstract basis, which approximate no other element, may be deleted without changing the ideal completion.
26. Show that if  $x$  is a non-compact element of a basis  $B$  for a continuous domain  $D$  then  $B \setminus \{x\}$  is still a basis. (Hint: Use the interpolation property.)
27. The preceding exercise shows that different bases can generate the same domain. Show that for a fixed basis different orders of approximation may also yield the same domain. Show that this will definitely be the case if the two orders  $\prec_1$  and  $\prec_2$  satisfy the equations  $\prec_1 \circ \prec_2 = \prec_1$  and  $\prec_2 \circ \prec_1 = \prec_2$ .
28. What is the ideal completion of  $(\mathbb{Q}, <)$ ?
29. Let  $\prec$  be a relation on a set  $B$  such that  $\prec \circ \prec = \prec$  holds. Give an example showing that Axiom (INT) (Definition 2.2.20) need not be satisfied. Nevertheless,  $\text{Idl}(B, \prec)$  is a continuous domain. What is the advantage of our axiomatization over this simpler concept?
30. Spell out the proof of Theorem 2.2.28.
31. Prove that in a dcpo every upper set is the intersection of its Scott-neighbourhoods.
32. Show that in order to construct the Scott-closure of a lower set  $A$  of a continuous domain it is sufficient to add all suprema of directed subsets to  $\downarrow A$ . Give an example of a non-continuous dcpo where this fails.

33. Given a subset  $X$  in a dcpo  $D$  let  $\bar{X}$  be the smallest superset of  $X$  which is closed against the formation of suprema of directed subsets. Show that the cardinality of  $\bar{X}$  can be no greater than  $2^{|X|}$ . (Hint: Construct a directed suprema closed superset of  $X$  by adding *all* existing suprema to  $X$ .)

### 3 Domains collectively

#### 3.1 Comparing domains

##### 3.1.1 Retractions

A reader with some background in universal algebra may already have missed a discussion of sub-dcpo's and quotient-dcpo's. The reason for this omission is quite simple: there is no fully satisfactory notion of sub-object or quotient in domain theory based on general Scott-continuous functions. And this is because the formation of directed suprema is a partial operation of unbounded arity. We therefore cannot hope to be able to employ the tools of universal algebra. But if we *combine* the ideas of sub-object and quotient then the picture looks quite nice.

**Definition 3.1.1.** Let  $P$  and  $Q$  be posets. A pair  $s: P \rightarrow Q, r: Q \rightarrow P$  of monotone functions is called a *monotone section retraction pair* if  $r \circ s$  is the identity on  $P$ . In this situation we will call  $P$  a *monotone retract* of  $Q$ .

If  $P$  and  $Q$  are dcpo's and if both functions are continuous then we speak of a *continuous section retraction pair*.

We will omit the qualifying adjective 'monotone', respectively 'continuous', if the properties of the functions are clear from the context. We will also use *s-r-pair* as a shorthand.

One sees immediately that in an s-r-pair the retraction is surjective and the section is injective, so our intuition about  $P$  being both a sub-object and a quotient of  $Q$  is justified. In such a situation  $P$  inherits many properties from  $Q$ :

**Proposition 3.1.2.** *Let  $P$  and  $Q$  be posets and let  $s: P \rightarrow Q, r: Q \rightarrow P$  be a monotone section retraction pair.*

1. *Let  $A$  be any subset of  $P$ . If  $s(A)$  has a supremum in  $Q$  then  $A$  has a supremum in  $P$ . It is given by  $r(\bigsqcup s(A))$ . Similarly for the infimum.*
2. *If  $Q$  is a (pointed) dcpo, a semilattice, a lattice or a complete lattice then so is  $P$ .*

**Proof.** Because of  $r \circ s = \text{id}_P$  and the monotonicity of  $r$  it is clear that  $r(\bigsqcup s(A))$  is an upper bound for  $A$ . Let  $x$  be another such. Then by the monotonicity of  $s$  we have that  $s(x)$  is an upper bound of  $s(A)$  and hence it is above  $\bigsqcup s(A)$ . So we get  $x = r(s(x)) \sqsupseteq r(\bigsqcup s(A))$ .

The property of being a (pointed) dcpo, semilattice, etc., is defined through the existence of suprema or infima of certain subsets. The shape

of these subsets is preserved by monotone functions and so (2) follows from (1). ■

Let us now turn to continuous section retraction pairs.

**Lemma 3.1.3.** *Let  $(s, r)$  be a continuous section retraction pair between dcpos  $D$  and  $E$  and let  $B$  be a basis for  $E$ . Then  $r(B)$  is a basis for  $D$ .*

**Proof.** Let  $c \in B$  be an approximant to  $s(x)$  for  $x \in D$ . We show that  $r(c)$  approximates  $x$ . To this end let  $A$  be a directed subset of  $D$  with  $\bigcup^\uparrow A \sqsupseteq x$ . By the continuity of  $s$  we have  $\bigcup^\uparrow s(A) = s(\bigcup^\uparrow A) \sqsupseteq s(x)$  and so for some  $a \in A$ ,  $s(a) \sqsupseteq c$  must hold. This implies  $a = r(s(a)) \sqsupseteq r(c)$ . The continuity of  $r$  gives us that  $x$  is the supremum of  $r(B_{s(x)})$ . ■

**Theorem 3.1.4.** *A retract of a continuous domain via a continuous s-r-pair is continuous.*

The analogous statement for algebraic domains does not hold in general. Instead of constructing a particular counterexample, we use our knowledge about the ideal completion to get a general, positive result which implies this negative one.

**Theorem 3.1.5.** *Every  $(\omega)$ -continuous domain is the retract of an  $(\omega)$ -algebraic domain via a continuous s-r-pair.*

In more detail, we have:

**Proposition 3.1.6.** *Let  $D$  be a continuous domain with basis  $B$ . Then the maps  $s: D \rightarrow \text{Idl}(B, \sqsubseteq), x \mapsto B_x$  and  $r: \text{Idl}(B, \sqsubseteq) \rightarrow D, A \mapsto \bigcup^\uparrow A$  constitute a continuous section retraction pair between  $D$  and  $\text{Idl}(B, \sqsubseteq)$ .*

**Proof.** The continuity of  $r$  follows from general associativity, Proposition 2.1.4, and the fact that directed suprema in  $\text{Idl}(B)$  are directed unions. For the continuity of  $s$  we use the interpolation property in the form of Proposition 2.2.16(2). ■

### 3.1.2 Idempotents

Often the section part of an s-r-pair is really a subset inclusion. In this case we can hide it and work with the map  $s \circ r$  on  $E$  alone. It is idempotent, because  $(s \circ r) \circ (s \circ r) = s \circ (r \circ s) \circ r = s \circ r$ .

**Proposition 3.1.7.**

1. *The image of a continuous idempotent map  $f$  on a dcpo  $D$  is a dcpo. The suprema of directed subsets of  $\text{im}(f)$ , calculated in  $\text{im}(f)$ , coincide with those calculated in  $D$ . The inclusion  $\text{im}(f) \rightarrow D$  is Scott-continuous.*
2. *The set of all continuous idempotent functions on a dcpo is again a dcpo.*

**Proof.** (1) The first part follows from Proposition 3.1.2 because the inclusion is surely monotone. For the second part let  $A$  be a directed set

contained in  $\text{im}(f)$ . We need to see that  $\bigsqcup^\uparrow A$  belongs to  $\text{im}(f)$  again. This holds because  $f$  is continuous:  $\bigsqcup^\uparrow A = \bigsqcup^\uparrow f(A) = f(\bigsqcup^\uparrow A)$ .

(2) Let  $(f_i)_{i \in I}$  be a directed family of continuous idempotents. For any  $x \in D$  we can calculate

$$\begin{aligned}
 (\bigsqcup_{i \in I}^\uparrow f_i) \circ (\bigsqcup_{j \in I}^\uparrow f_j)(x) &= \bigsqcup_{i \in I}^\uparrow f_i(\bigsqcup_{j \in I}^\uparrow f_j(x)) \\
 &= \bigsqcup_{i \in I}^\uparrow \bigsqcup_{j \in I}^\uparrow f_i(f_j(x)) \\
 &= \bigsqcup_{i \in I}^\uparrow f_i(f_i(x)) \\
 &= \bigsqcup_{i \in I}^\uparrow f_i(x).
 \end{aligned}$$

Hence the supremum of continuous idempotents is again an idempotent function. We have proved in Proposition 2.1.18 that it is also continuous. ■

If  $f$  is a continuous idempotent map on a continuous domain  $D$  then we know that its image is again continuous. But it is *not* true that the order of approximation on  $\text{im}(f)$  is the restriction of the order of approximation on  $D$ . For example, every constant map is continuous and idempotent. Its image is an algebraic domain with one element, which is therefore compact. But surely not every element of a continuous domain is compact. However, we can say something nice about the Scott-topology on the image:

**Proposition 3.1.8.** *If  $f$  is a continuous idempotent function on a dcpo  $D$  then the Scott-topology on  $\text{im}(f)$  is the restriction of the Scott-topology on  $D$  to  $\text{im}(f)$ .*

**Proof.** This follows immediately because a continuous idempotent function  $f$  gives rise to a continuous s-r-pair between  $\text{im}(f)$  and  $D$ . ■

Useful examples of idempotent self-maps are retractions  $\text{ret}_x$  onto principal ideals. They are given by

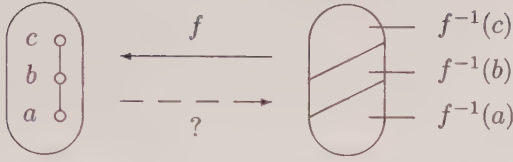
$$\text{ret}_x(y) = \begin{cases} y, & \text{if } y \sqsubseteq x; \\ x, & \text{otherwise.} \end{cases}$$

Their continuity follows from the fact that  $\downarrow x$  is always Scott-closed. Dually, we can define a retraction onto a principal filter  $\uparrow c$ . It is Scott-continuous if its generator  $c$  is compact.

### 3.1.3 Adjunctions

An easy way to avoid writing this subsection would be to refer to category theory and to translate the general theory of adjoint functors into the poset setting. However, we feel that the right way to get used to the idea





**Fig. 7.** The right inverse problem for a surjective function

of adjointness is to start out with a relatively simple situation such as is presented by domain theory. (In fact, we will use adjoint functors later on, but really in a descriptive fashion only.)

Let us start with the example of a surjective map  $f$  from a poset  $Q$  onto a poset  $P$ . It is natural to ask whether there is a one-sided inverse  $e: P \rightarrow Q$  for  $f$ , i.e. a map such that  $f \circ e = \text{id}_P$  holds. Figure 7 illustrates this situation. Such a map must pick out a representative from  $f^{-1}(x)$  for each  $x \in P$ . Set-theoretically this can be done, but the point here is that we want  $e$  to be monotone. If we succeed then  $e$  and  $f$  form a (monotone) section retraction pair. Even nicer would it be if we could pick out a canonical representative from  $f^{-1}(x)$ , which in the realm of order theory means that we want  $f^{-1}(x)$  to have a least (or largest) element. If this is the case then how can we ensure that the assignment  $e: x \mapsto \min(f^{-1}(x))$  is monotone? The solution is suggested by the observation that if  $e$  is monotone then  $e(x)$  is the least element not only of  $f^{-1}(x)$  but also of  $f^{-1}(\uparrow x)$ . This condition is also sufficient. The switch from  $f^{-1}(x)$  to  $f^{-1}(\uparrow x)$  (and this is a trick to remember) may allow us to construct a partial right inverse even if  $f$  is not surjective. Thus we arrive at a first, tentative definition of an adjunction.

**Definition 3.1.9.** (Preliminary) Let  $P$  and  $Q$  be posets and let  $l: P \rightarrow Q$  and  $u: Q \rightarrow P$  be monotone functions. We say that  $(l, u)$  is an *adjunction* between  $P$  and  $Q$  if for every  $x \in P$  we have that  $l(x)$  is the least element of  $u^{-1}(\uparrow x)$ .

This definition is simple and easy to motivate. But it brings out just one aspect of adjoint pairs, namely, that  $l$  is uniquely determined by  $u$ . There is much more:

**Proposition 3.1.10.** Let  $P$  and  $Q$  be posets and  $l: P \rightarrow Q$  and  $u: Q \rightarrow P$  be monotone functions. Then the following are equivalent:

1.  $\forall x \in P. l(x) = \min(u^{-1}(\uparrow x))$ ,
2.  $\forall y \in Q. u(y) = \max(l^{-1}(\downarrow y))$ ,
3.  $l \circ u \sqsubseteq \text{id}_Q$  and  $u \circ l \sqsupseteq \text{id}_P$ ,
4.  $\forall x \in P \forall y \in Q. (x \sqsubseteq u(y) \Leftrightarrow l(x) \sqsubseteq y)$ .

(For (4) $\Rightarrow$ (1) the monotonicity of  $u$  and  $l$  is not needed.)

**Proof.** (1) $\implies$ (2) Pick an element  $y \in Q$ . Then because  $u(y) \sqsubseteq u(y)$  we have from (1) that  $l(u(y)) \sqsubseteq y$  holds. So  $u(y)$  belongs to  $l^{-1}(\downarrow y)$ . Now let  $x'$  be any element of  $l^{-1}(\downarrow y)$ , or, equivalently,  $l(x') \sqsubseteq y$ . Using (1) again, we see that this can only happen if  $u(y) \sqsupseteq x'$  holds. So  $u(y)$  is indeed the largest element of  $l^{-1}(\downarrow y)$ . The converse is proved analogously, of course.

(1) and (2) together immediately give both (3) and (4).

From (3) we get (4) by applying the monotone map  $l$  to the inequality  $x \sqsubseteq u(y)$  and using  $l \circ u \sqsubseteq \text{id}_Q$ .

Assuming (4) we see immediately that  $l(x)$  is a lower bound for  $u^{-1}(\uparrow x)$ . But because  $l(x) \sqsubseteq l(x)$  and hence  $x \sqsubseteq u(l(x))$  we have that  $l(x)$  also belongs to  $u^{-1}(\uparrow x)$ . We get the monotonicity of  $l$  as follows: If  $x \sqsubseteq x'$  holds in  $P$  then because  $l(x') \sqsubseteq l(x')$  we have  $x' \sqsubseteq u(l(x'))$  and by transitivity  $x \sqsubseteq u(l(x'))$ . Using (4) again, we get  $l(x) \sqsubseteq l(x')$ .  $\blacksquare$

We conclude that despite the lopsided definition, the situation described by an adjunction is completely symmetric. And indeed, adjunctions are usually introduced using either (3) or (4).

**Definition 3.1.11.** (*Official*) Let  $P$  and  $Q$  be posets and let  $l: P \rightarrow Q$  and  $u: Q \rightarrow P$  be functions. We say that  $(l, u)$  is an *adjunction* between  $P$  and  $Q$  if for all  $x \in P$  and  $y \in Q$  we have  $x \sqsubseteq u(y) \Leftrightarrow l(x) \sqsubseteq y$ . We call  $l$  the *lower* and  $u$  the *upper adjoint* and write  $l: P \rightleftarrows Q: u$ .

**Proposition 3.1.12.** *Let  $l: P \rightleftarrows Q: u$  be an adjunction between posets.*

1.  $u \circ l \circ u = u$  and  $l \circ u \circ l = l$ ,
2. *The image of  $u$  and the image of  $l$  are order-isomorphic. The isomorphisms are given by the restrictions of  $u$  and  $l$  to  $\text{im}(l)$  and  $\text{im}(u)$ , respectively.*
3.  $u$  is surjective  $\Leftrightarrow u \circ l = \text{id}_P \Leftrightarrow l$  is injective,
4.  $l$  is surjective  $\Leftrightarrow l \circ u = \text{id}_Q \Leftrightarrow u$  is injective,
5.  $l$  preserves existing suprema,  $u$  preserves existing infima.

**Proof.** (1) We use Proposition 3.1.10(3) twice:  $u = \text{id}_P \circ u \sqsubseteq (u \circ l) \circ u = u \circ (l \circ u) \sqsubseteq u \circ \text{id}_Q = u$ .

(2) The equations from (1) say precisely that on the images of  $u$  and  $l$ ,  $u \circ l$  and  $l \circ u$ , respectively, act like identity functions.

(3) If  $u$  is surjective then we can cancel  $u$  on the right in the equation  $u \circ l \circ u = u$  and get  $u \circ l = \text{id}_P$ . From this it follows that  $l$  must be injective.

(5) Let  $x = \bigsqcup A$  for  $A \subseteq P$ . By monotonicity,  $l(x) \sqsupseteq l(a)$  for each  $a \in A$ . Conversely, let  $y$  be any upper bound of  $l(A)$ . Then  $u(y)$  is an upper bound for each  $u(l(a))$  which in turn is above  $a$ . So  $u(y) \sqsupseteq \bigsqcup A = x$  holds and this is equivalent to  $y \sqsupseteq l(x)$ .  $\blacksquare$

The last property in the preceding proposition may be used to define an adjunction in yet another way, the only prerequisite being that there are

enough sets with an infimum (or supremum). This is the adjoint functor theorem for posets.

**Proposition 3.1.13.** *Let  $f: L \rightarrow P$  be a monotone function from a complete lattice to a poset. Then the following are equivalent:*

1.  $f$  preserves all infima,
2.  $f$  has a lower adjoint.

And similarly:  $f$  preserves all suprema if and only if  $f$  has an upper adjoint.

**Proof.** We already know how to define a candidate for a lower adjoint  $g$ ; we try  $g(x) = \sqcap f^{-1}(\uparrow x)$ . All that remains is to show that  $g(x)$  belongs to  $f^{-1}(\uparrow x)$ . This follows because  $f$  preserves meets:  $f(g(x)) = f(\sqcap f^{-1}(\uparrow x)) = \sqcap f(f^{-1}(\uparrow x)) \sqsupseteq \sqcap \uparrow x = x$ . ■

This proposition gives us a way of recognizing an adjoint situation in cases where only one function is explicitly given. It is then useful to have a notation for the missing mapping. We write  $f^*$  for the upper and  $f_*$  for the lower adjoint of  $f$ .

Now it is high time to come back to domains and see what all this means in our setting.

**Proposition 3.1.14.** *Let  $l: D \rightleftharpoons E: u$  be an adjunction between dcpos.*

1.  $l$  is Scott-continuous.
2. If  $u$  is Scott-continuous then  $l$  preserves the order of approximation.
3. If  $D$  is continuous then the converse of (2) is also true.

**Proof.** Continuity of the lower adjoint follows from Proposition 3.1.12(5). So let  $x \ll y$  be elements in  $D$  and let  $A$  be a directed subset of  $E$  such that  $l(y) \sqsubseteq \sqcup^1 A$  holds. This implies  $y \sqsubseteq u(\sqcup^1 A)$  and from the continuity of  $u$  we deduce  $y \sqsubseteq \sqcup^1 u(A)$ . Hence some  $u(a)$  is above  $x$  which, going back to  $E$ , means  $l(x) \sqsubseteq a$ .

(3) For the converse let  $A$  be any directed subset of  $E$ . Monotonicity of  $u$  yields  $\sqcup^1 u(A) \sqsubseteq u(\sqcup^1 A)$ . In order to show that the other inequality also holds, we prove that  $\sqcup^1 u(A)$  is above every approximant to  $u(\sqcup^1 A)$ . Indeed, if  $x \ll u(\sqcup^1 A)$  we have  $l(x) \ll l(u(\sqcup^1 A)) \sqsubseteq \sqcup^1 A$  by assumption. So some  $a$  is above  $l(x)$  and for this  $a$  we have  $x \sqsubseteq u(a) \sqsubseteq \sqcup^1 u(A)$ . ■

### 3.1.4 Projections and sub-domains

Let us now combine the ideas of Section 3.1.1 and 3.1.3.

**Definition 3.1.15.** Let  $D$  and  $E$  be dcpos and let  $e: D \rightarrow E$  and  $p: E \rightarrow D$  be continuous functions. We say that  $(e, p)$  is a *continuous embedding projection pair* (or *e-p-pair*) if  $p \circ e = \text{id}_D$  and  $e \circ p \sqsubseteq \text{id}_E$ .

We note that the section retraction pair between a continuous domain and its ideal completion as constructed in Section 3.1.1 is really an embedding projection pair.

From the general theory of adjunctions and retractions we already know quite a bit about e-p-pairs. The embedding is injective,  $p$  is surjective,  $e$  preserves existing suprema and the order of approximation,  $p$  preserves existing infima,  $D$  is continuous if  $E$  is continuous, and, finally, embeddings and projections uniquely determine each other. Because of this last property the term ‘embedding’ has a well-defined meaning; it is an injective function which has a Scott-continuous upper adjoint.

An injective lower adjoint also reflects the order of approximation:

**Proposition 3.1.16.** *Let  $e:D \rightleftharpoons E:p$  be an e-p-pair between dcpos and let  $x$  and  $y$  be elements of  $D$ . Then  $e(x) \ll e(y)$  holds in  $E$  if and only if  $x$  approximates  $y$  in  $D$ .*

Let us also look at the associated idempotent  $e \circ p$  on  $E$ . As it is below the identity, it makes good sense to call such a function a *kernel operator*, but often such maps are just called *projections*. We denote the set of kernel operators on a dcpo  $D$  by  $[D \overset{\downarrow}{\rightarrow} D]$ . It is important to note that while a kernel operator preserves infima as a map from  $D$  to its image, it does *not* have any preservation properties as a map from  $D$  to  $D$  besides Scott-continuity. What we can say is summarized in the following proposition.

**Proposition 3.1.17.** *Let  $D$  be a dcpo.*

1.  $[D \overset{\downarrow}{\rightarrow} D]$  is a dcpo.
2. If  $p$  is a kernel operator on  $D$  then for all  $x \in D$  we have that  $p(x) = \max\{y \in \text{im}(p) \mid y \sqsubseteq x\}$ .
3. The image of a kernel operator is closed under existing suprema.
4.  $\ll_{\text{im}(p)} = (\ll_D) \cap (\text{im}(p) \times \text{im}(p))$ .
5. For kernel operators  $p, p'$  on  $D$  we have  $p \sqsubseteq p'$  if and only if  $\text{im}(p) \subseteq \text{im}(p')$ .

**Proof.** (1) is proved as Proposition 3.1.7 and (2) follows because  $p$  together with the inclusion of  $\text{im}(p)$  into  $D$  form an adjunction. This also shows (4). Finally, (3) and (5) are direct consequences of (2). ■

In the introduction we explained the idea that the order on a semantic domain models the relation of some elements being *better* than others, where—at least in semantics—‘better’ may be replaced more precisely by ‘better termination’. Thus we view elements at the bottom of a domain as being less desirable than those higher up; they are ‘proto-elements’ from which fully developed elements evolve as we go up in the order. Now, the embedding part of an e-p-pair  $e:D \rightleftharpoons E:p$  places  $D$  at the bottom of  $E$ . Following the above line of thought, we may think of  $D$  as being a collection of proto-elements from which the elements of  $E$  evolve. Because there is the projection part as well, every element of  $E$  exists in some primitive form in  $D$  already. Also,  $D$  contains some information about the order



and the order of approximation on  $E$ . We may therefore think of  $D$  as a preliminary version of  $E$ , as an *approximation* to  $E$  on the domain level. This thought is made fruitful in Sections 4.2 and 5. Although the word does not convey the whole fabric of ideas, we name  $D$  a *sub-domain* of  $E$ , just in case there is an e-p-pair  $e: D \rightleftharpoons E:p$ .

### 3.1.5 Closures and quotient domains

The sub-domain relation is pre-eminent in domain theory but, of course, we can also combine retractions and adjunctions the other way around. Thus we arrive at *continuous insertion closure pairs* (*i-c-pairs*). Because adjunctions are not symmetric as far as the order of approximation is concerned, Proposition 3.1.14, the situation is not just the order dual of that of the previous subsection. We know that the insertion preserves existing infima and so on, but in addition we now have that the surjective part preserves the order of approximation, and therefore  $D$  is algebraic if  $E$  is.

The associated idempotent is called a *closure operator*. For closure operators the same caveat applies as for kernel operators; they need not preserve suprema. Worse, such functions no longer automatically have a Scott-continuous (upper) adjoint. This is the price we have to pay for the algebraicity of the image. Let us formulate this precisely.

**Proposition 3.1.18.** *Let  $D$  be an algebraic domain and let  $c: D \rightarrow D$  be a monotone idempotent function above  $\text{id}_D$ . Then  $\text{im}(c)$  is again an algebraic domain if and only if it is closed under directed suprema.*

The reader will no doubt recognize this statement as being a reformulation and generalization of our example of inductive closure systems from Proposition 2.2.9. It is only consequent to call  $D$  a *quotient domain* of the continuous domain  $E$  if there exists an i-c-pair  $e: D \rightleftharpoons E:c$ .

## 3.2 Finitary constructions

In this section we will present a few basic ways of putting domains together so as to build up complicated structures from simple ones. There are three aspects of these constructions which we are interested in. The first one is simply the order-theoretic definition and the proof that we stay within dcpos and Scott-continuous functions. The second one is the question how the construction can be described in terms of bases and whether the principle of approximation can be retained. The third one, finally, is the question of what universal property the construction has. This is the categorical viewpoint. Since Vol. 1 contains a chapter on category theory, [Poigné, 1992] (in particular, Chapter 2), we need not repeat here the arguments for why this is a fruitful and enlightening way of looking at these type constructors.

There are, however, several categories that we are interested in and a construction may play different roles in different settings. Let us there-



fore list the categories that, at this point, seem suitable as a *universe of discourse*. There is, first of all, **DCPO**, the category of dcpos and Scott-continuous functions as introduced in Section 2.1. We can restrict the objects by taking only continuous or, more special, algebraic domains. Thus we arrive at the full subcategories **CONT** and **ALG** of **DCPO**. Each of these may be further restricted by requiring the objects to have a bottom element (and Theorem 2.1.19 tells us why one would be interested in doing so) resulting in the categories **DCPO**<sub>⊥</sub>, **CONT**<sub>⊥</sub>, and **ALG**<sub>⊥</sub>. The presence of a distinguished point in each object suggests that morphisms should preserve them. But this is not really appropriate in semantics; strict functions are tied to a particular evaluation strategy. For us this means that there is yet another cascade of categories, **DCPO**<sub>⊥!</sub>, **CONT**<sub>⊥!</sub>, and **ALG**<sub>⊥!</sub>, where objects have bottom elements and morphisms are strict and Scott-continuous. Finally, we may bound the size of (minimal) bases for continuous and algebraic domains to be countable. We indicate this by the prefix ‘ $\omega$ -’.

### 3.2.1 Cartesian product

**Definition 3.2.1.** The *cartesian product* of two dcpos  $D$  and  $E$  is given by the following data:

$$\begin{aligned} D \times E &= \{\langle x, y \rangle \mid x \in D, y \in E\}, \\ \langle x, y \rangle &\sqsubseteq \langle x', y' \rangle \text{ if and only if } x \sqsubseteq x' \text{ and } y \sqsubseteq y'. \end{aligned}$$

This is just the usual product of sets, augmented by the coordinatewise order. Through induction, we can define the cartesian product for finite non-empty collections of dcpos. For the product over the empty index set we define the result to be a fixed one-element dcpo  $\mathbb{I}$ .

**Proposition 3.2.2.** *The cartesian product of dcpos is a dcpo. Suprema and infima are calculated coordinatewise.*

With each product  $D \times E$  there are associated two projections:

$$\pi_1: D \times E \rightarrow D \text{ and } \pi_2: D \times E \rightarrow E.$$

These projections are always surjective but they are upper adjoints only if  $D$  and  $E$  are pointed. So there is a slight mismatch with Section 3.1.4 here. Given a dcpo  $F$  and continuous functions  $f: F \rightarrow D$  and  $g: F \rightarrow E$ , we denote the mediating morphism from  $F$  to  $D \times E$  by  $\langle f, g \rangle$ . It maps  $x \in F$  to  $\langle f(x), g(x) \rangle$ .

**Proposition 3.2.3.** *Projections and mediating morphisms are continuous.*

If  $f: D \rightarrow D'$  and  $g: E \rightarrow E'$  are Scott-continuous, then so is the mediating map  $\langle f \circ \pi_1, g \circ \pi_2 \rangle: D \times E \rightarrow D' \times E'$ . The common notation for it

is  $f \times g$ . Since our construction is completely explicit, we have thus defined a functor in two variables on **DCPO**.

**Proposition 3.2.4.** *Let  $D$  and  $E$  be dcpos.*

1. *A tuple  $\langle x, y \rangle$  approximates a tuple  $\langle x', y' \rangle$  in  $D \times E$  if and only if  $x$  approximates  $x'$  in  $D$  and  $y$  approximates  $y'$  in  $E$ .*
2. *If  $B$  and  $B'$  are bases for  $D$  and  $E$ , respectively, then  $B \times B'$  is a basis for  $D \times E$ .*
3.  *$D \times E$  is continuous if and only if  $D$  and  $E$  are.*
4.  *$K(D \times E) = K(D) \times K(E)$ .*

The categorical aspect of the cartesian product is quite pleasing; it is a categorical product in each case. But we can say even more.

**Lemma 3.2.5.** *Let  $\mathbf{C}$  be a full subcategory of **DCPO** or **DCPO**<sub>⊥</sub> which has finite products. Then these are isomorphic to the cartesian product.*

In a restricted setting this was first observed in [Smyth, 1983a]. The general proof may be found in [Jung, 1989].

A useful property, which does not follow from general categorical or topological considerations, is the following.

**Lemma 3.2.6.** *A function  $f: D \times E \rightarrow F$  is continuous if and only if it is continuous in each variable separately.*

**Proof.** Assume  $f: D \times E \rightarrow F$  is separately continuous. Then  $f$  is monotone, because given  $(x, y) \sqsubseteq (x', y')$  we can fill in  $(x, y')$  and use coordinatewise monotonicity twice. The same works for continuity: if  $A \subseteq D \times E$  is directed then

$$\begin{aligned}
 \bigsqcup_{(x,y) \in A}^{\uparrow} f(x, y) &= \bigsqcup_{x \in \pi_1(A)}^{\uparrow} \bigsqcup_{y \in \pi_2(A)}^{\uparrow} f(x, y) \\
 &= \bigsqcup_{x \in \pi_1(A)}^{\uparrow} f(x, \bigsqcup_{y \in \pi_2(A)}^{\uparrow} y) \\
 &= f(\bigsqcup_{x \in \pi_1(A)}^{\uparrow} x, \bigsqcup_{y \in \pi_2(A)}^{\uparrow} y) \\
 &= f(\bigsqcup^{\uparrow} A).
 \end{aligned}$$

This proves the interesting direction. ■

### 3.2.2 Function space

We have introduced the function space in Section 2.1.6 already. It consists of all continuous functions between two dcpos ordered pointwise. We know that this is again a dcpo. The first morphism which is connected with this construction is  $\text{apply}: [D \rightarrow E] \times D \rightarrow E$ ,  $\langle f, x \rangle \mapsto f(x)$ . It is continuous because it is continuous in each argument separately: in the

first because directed suprema of functions are calculated pointwise, in the second, because  $[D \rightarrow E]$  contains only continuous functions.

The second standard morphism is the operation which rearranges a function of two arguments into a combination of two unary functions. That is, if  $f$  maps  $D \times E$  to  $F$ , then  $\text{Curry}(f): D \rightarrow [E \rightarrow F]$  is the mapping which assigns to  $d \in D$  the function which assigns to  $e \in E$  the element  $f(d, e)$ .  $\text{Curry}(f)$  is a continuous function because of Lemma 3.2.6. And for completely general reasons we have that  $\text{Curry}$  itself is a continuous operation from  $[D \times E \rightarrow F]$  to  $[D \rightarrow [E \rightarrow F]]$ . Another derived operation is composition, which is a continuous operation from  $[D \rightarrow E] \times [E \rightarrow F]$  to  $[D \rightarrow F]$ .

All this shows that the continuous function space is the exponential in **DCPO**. Taking cartesian products and function spaces together we have shown that **DCPO** is cartesian closed.

We turn the function space construction into a functor from  $\mathbf{DCPO}^{op} \times \mathbf{DCPO}$  to **DCPO** by setting  $[\cdot \rightarrow \cdot](f, g)(h) = g \circ h \circ f$ , where  $f: D' \rightarrow D$ ,  $g: E \rightarrow E'$  and  $h$  is an element of  $[D \rightarrow E]$ .

As for the product we can show that the choice of the exponential object is more or less forced upon us. This again was first noticed by Smyth in the above-mentioned reference.

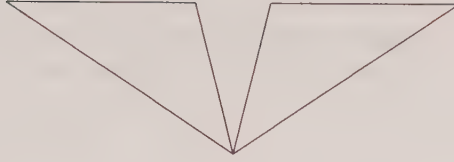
**Lemma 3.2.7.** *Let  $\mathbf{C}$  be a cartesian closed full subcategory of **DCPO**. The exponential of two objects  $D$  and  $E$  of  $\mathbf{C}$  is isomorphic to  $[D \rightarrow E]$ .*

Let us now turn to the theme of approximation in function spaces. The reader should brace himself for a profound disappointment: Even for algebraic domains it may be the case that the order of approximation on the function space is empty! (Exercise 3.3.12(11) discusses an example.) This fact together with Lemmas 3.2.5 and 3.2.7 implies that neither **CONT** nor **ALG** are cartesian closed. The only way to remedy this situation is to move to more restricted classes of domains. This will be the topic of Chapter 4.

### 3.2.3 Coalesced sum

In the category of sets the coproduct is given by disjoint union. This works equally well for dcpos and there is not really anything interesting to prove about it. We denote it by  $D \dot{\cup} E$ .

Disjoint union, however, destroys the property of having a least element and this in turn is indispensable in proving that every function has a fixpoint, Theorem 2.1.19. One therefore looks for substitutes for disjoint union which retain pointedness, but, of course, one cannot expect a clean categorical characterization such as for cartesian product or function space. (See also Exercise 3.3.12(12).) In fact, it has been shown in [Huwig and Poigné, 1990] that we cannot have cartesian closure, the fixpoint property and coproducts in a non-degenerate category.



**Fig. 8.** The coalesced sum of two pointed dcpos.

So let us now restrict attention to pointed dcpos. One way of putting a family of them together is to identify their bottom elements. This is called the *coalesced sum* and denoted  $D \oplus E$ . Figure 8 illustrates this operation. Elements from  $D \oplus E$  different from  $\perp_{D \oplus E}$  carry a label which indicates where they came from. We write them in the form  $(x:i)$ ,  $i \in \{1, 2\}$ .

**Proposition 3.2.8.** *The coalesced sum of pointed dcpos is a pointed dcpo.*

The individual dcpos may be injected into the sum in the obvious way:

$$\text{inl}(x) = \begin{cases} (x:1), & x \neq \perp_D; \\ \perp_{D \oplus E}, & x = \perp_D; \end{cases}$$

and

$$\text{inr}(x) = \begin{cases} (x:2), & x \neq \perp_E; \\ \perp_{D \oplus E}, & x = \perp_E. \end{cases}$$

A universal property for the sum holds only in the realm of strict functions:

**Proposition 3.2.9.** *The coalesced sum of pointed dcpos is the coproduct in  $\mathbf{DCPO}_{\perp!}$ ,  $\mathbf{CONT}_{\perp!}$ , and  $\mathbf{ALG}_{\perp!}$ .*

Once we accept the restriction to bottom preserving functions it is clear how to turn the coalesced sum into a functor.

### 3.2.4 Smash product and strict function space

It is clear that inside  $\mathbf{DCPO}_{\perp!}$  a candidate for the exponential is not the full function space but rather the set  $[D \xrightarrow{\perp!} E]$  of strict continuous functions from  $D$  to  $E$ . However, it does not harmonize with the product in  $\mathbf{DCPO}_{\perp!}$ , which, as we have seen, must be the cartesian product. We do get a match if we consider the so-called *smash product*. It is defined like the cartesian product but all tuples which contain at least one bottom element are identified. Common notation is  $D \otimes E$ .

We leave it to the reader to check that smash product and strict function space turn  $\mathbf{DCPO}_{\perp!}$  into a monoidal closed category.

### 3.2.5 Lifting

Set-theoretically, *lifting* is the simple operation of adding a new bottom element to a dcpo. Applied to  $D$ , the resulting structure is denoted by  $D_{\perp}$ . Clearly, continuity or algebraicity don't suffer any harm from this.

Associated with it is the map  $\text{up}: D \rightarrow D_{\perp}$  which maps each  $x \in D$  to its namesake in  $D_{\perp}$ .

The categorical significance of lifting stems from the fact that it is left adjoint to the inclusion functor from  $\mathbf{DCPO}_{\perp!}$  into  $\mathbf{DCPO}$ . (Where a morphism  $f: D \rightarrow E$  is lifted by mapping the new bottom element of  $D_{\perp}$  to the new bottom element of  $E_{\perp}$ .)

### 3.2.6 Summary

For quick reference let us compile a table of the constructions looked at so far (Table 1). A '✓' indicates that the category is closed under the respective construction, a '+' says that, in addition, the construction plays the expected categorical role as a product, exponential or coproduct, respectively. Observe that for the constructions considered in this section it makes no difference if we restrict the size of a (minimal) basis.

	DCPO	DCPO <sub>⊥</sub>	DCPO <sub>⊥!</sub>	CONT ALG	CONT <sub>⊥</sub> ALG <sub>⊥</sub>	CONT <sub>⊥!</sub> ALG <sub>⊥!</sub>
$D \times E$	+	+	+	+	+	+
$[D \rightarrow E]$	+	+	✓			
$D \dot{\cup} E$	+			+		
$D \otimes E$		✓	✓		✓	✓
$[D \xrightarrow{\perp!} E]$		✓	+			
$D \oplus E$		✓	+		✓	+
$D_{\perp}$	✓	✓	✓	✓	✓	✓

Table 1. Summary of constructions.

## 3.3 Infinitary constructions

The product and sum constructions from the previous section have infinitary counterparts. Generally, these work nicely as long as we are only concerned with questions of convergence, but they cause problems with respect to the order of approximation. This is exemplified by the fact that an infinite power of a finite poset may fail to be algebraic. In any case, there is not much use of these operations in semantics. Much more interesting is the idea of incrementally building up a domain in a limit process. This is the topic of this section.

### 3.3.1 Limits and colimits

Our limit constructions are to be understood categorically and hence we refer once more to [Poigné, 1992] for motivation and general results. Here



are the basic definitions. A *diagram* in a category  $\mathbf{C}$  is given by a functor from a small category  $\mathbf{I}$  to  $\mathbf{C}$ . We can describe, somewhat sloppily but more concretely, a diagram by a pair  $\langle (D_i)_{i \in O}, (f_j: D_{d(j)} \rightarrow D_{c(j)})_{j \in M} \rangle$  of a family of objects and a family of *connecting morphisms*. The shape of the diagram is thus encoded in the index sets  $O$  (which correspond to the objects of  $\mathbf{I}$ ) and  $M$  (which correspond to the morphisms of  $\mathbf{I}$ ) and in the maps  $c, d: M \rightarrow O$  which corresponds to the dom and codom map on  $\mathbf{I}$ . What is lost is the information about composition in  $\mathbf{I}$ . In the cases which interest us, this is not a problem. A *cone* over such a diagram is given by an object  $D$  and a family  $(f_i: D \rightarrow D_i)_{i \in O}$  of morphisms such that for all  $j \in M$  we have  $f_j \circ f_{d(j)} = f_{c(j)}$ . A cone is *limiting* if for every other cone  $\langle E, (g_i)_{i \in O} \rangle$  there is exactly one morphism  $f: E \rightarrow D$  such that for all  $i \in O$ ,  $g_i = f_i \circ f$ . If  $\langle D, (f_i)_{i \in O} \rangle$  is a limiting cone, then  $D$  is called *limit* and the  $f_i$  are called *limiting morphisms*. The dual notions are *cocone*, *colimit*, and *colimiting morphism*.

**Theorem 3.3.1.** *DCPO has limits of arbitrary diagrams.*

**Proof.** The proof follows general category theoretic principles. We describe the limit of the diagram  $\langle (D_i)_{i \in O}, (f_j: D_{d(j)} \rightarrow D_{c(j)})_{j \in M} \rangle$  as a set of particular elements of the product of all  $D_i$ s, the so-called *commuting tuples*:

$$D = \{ \langle x_i : i \in O \rangle \in \prod_{i \in O} D_i \mid \forall j \in M. x_{c(j)} = f_j(x_{d(j)}) \}.$$

The order on the limit object is inherited from the product, that is, tuples are ordered coordinatewise. It is again a dcpo because the coordinatewise supremum of commuting tuples is commuting as all  $f_j$  are Scott-continuous. This also proves that the projections  $\pi_j: \prod_{i \in O} D_i \rightarrow D_j$  restricted to  $D$  are continuous. They give us the maps needed to complement  $D$  to a cone.

Given any other cone  $\langle E, (g_i: E \rightarrow D_i)_{i \in O} \rangle$ , we define the mediating morphism  $h: E \rightarrow D$  by  $h(x) = \langle g_i(x) : i \in O \rangle$ . Again, it is obvious that this is well-defined and continuous, and that it is the only possible choice. ■

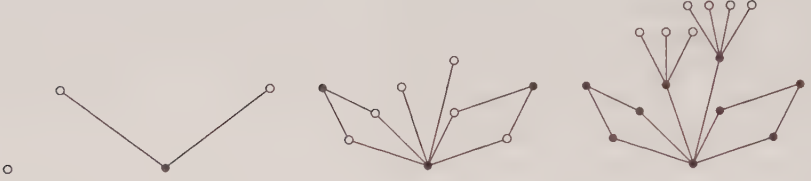
We also have the dual:

**Theorem 3.3.2.** *DCPO has colimits of arbitrary diagrams.*

This was first noted in [Markowsky, 1977] and, for a somewhat different setting, in [Meseguer, 1977]. The simplest way to prove it is by reducing it to completeness à la Theorem 23.14 of [Herrlich and Strecker, 1973]. This appears in [Lehmann and Smyth, 1981]. A more detailed analysis of colimits appears in [Fiech, 1992]. There the problem of retaining algebraicity is also addressed.

**Theorem 3.3.3.** *DCPO is cartesian closed, complete and cocomplete.*

**Theorem 3.3.4.** *DCPO<sub>⊥!</sub> is monoidal closed, complete and cocomplete.*



**Fig. 9.** An expanding sequence of finite domains.

How about  $\mathbf{DCPO}_\perp$ , where objects have least elements but morphisms need not preserve them? The truth is that both completeness and cocompleteness fail for this category. On the other hand, it is the right setting for denotational semantics in most cases. As a result of this mismatch, we quite often must resort to detailed proofs on the element level and cannot simply apply general category-theoretic principles. Let us nevertheless write down the one good property of  $\mathbf{DCPO}_\perp$ :

**Theorem 3.3.5.**  $\mathbf{DCPO}_\perp$  is cartesian closed.

### 3.3.2 The limit–colimit coincidence

The theorems of the previous subsection fall apart completely if we pass to domains, that is, to **CONT** or **ALG**. To get better results for limits and colimits we must restrict both the shape of the diagrams and the connecting morphisms used.

For motivation let us look at a chain  $D_1, D_2, \dots$  of domains where each  $D_n$  is a sub-domain of  $D_{n+1}$  in the sense of Section 3.1.4. Taking up again the animated language from that section we may think of the points of  $D_{n+1}$  as growing out of points of  $D_n$ , the latter being the buds which contain the leaves and flowers to be seen at later stages. Figure 9 shows, it is hoped, an inspiring example. Intuition suggests that in such a situation a well-structured limit can be found by adding limit points to the union of the  $D_n$ , and that it will be algebraic if the  $D_n$  are.

**Definition 3.3.6.** A diagram  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn}: D_n \rightarrow D_m)_{n \leq m \in \mathbb{N}} \rangle$  in the category  $\mathbf{DCPO}$  is called an *expanding sequence* if the following conditions are satisfied:

1. Each  $e_{mn}: D_n \rightarrow D_m$  is an embedding. (The associated projection  $e_{mn}^*$  we denote by  $p_{nm}$ .)
2.  $\forall n \in \mathbb{N}. e_{nn} = \text{id}_{D_n}$ .
3.  $\forall n \leq m \leq k \in \mathbb{N}. e_{kn} = e_{km} \circ e_{mn}$ .

Note that because lower adjoints determine upper adjoints and vice versa, we have  $p_{nk} = p_{nm} \circ p_{mk}$  whenever  $n \leq m \leq k \in \mathbb{N}$ .

It turns out that, in contrast to the general situation, the colimit of an expanding sequence can be calculated easily via the associated projections.

**Theorem 3.3.7.** *Let  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn} : D_n \rightarrow D_m)_{n \leq m \in \mathbb{N}} \rangle$  be an expanding sequence in **DCPO**. Define*

$$\begin{aligned} D &= \{ \langle x_n : n \in \mathbb{N} \rangle \in \prod_{n \in \mathbb{N}} D_n \mid \forall n \leq m \in \mathbb{N}. x_n = p_{nm}(x_m) \}, \\ p_m : D &\rightarrow D_m, \langle x_n : n \in \mathbb{N} \rangle \mapsto x_m, m \in \mathbb{N}, \\ e_m : D_m &\rightarrow D, x \mapsto \langle \bigsqcup_{k \sqsubseteq n, m}^\uparrow p_{nk} \circ e_{km}(x) : n \in \mathbb{N} \rangle, m \in \mathbb{N}. \end{aligned}$$

Then

1. The maps  $(e_m, p_m)$ ,  $m \in \mathbb{N}$ , form embedding projection pairs and  $\bigsqcup_{n \in \mathbb{N}}^\uparrow e_n \circ p_n = \text{id}_D$  holds.
2.  $\langle D, (p_n)_{n \in \mathbb{N}} \rangle$  is a limit of the diagram  $\langle (D_n)_{n \in \mathbb{N}}, (p_{nm})_{n \leq m \in \mathbb{N}} \rangle$ . If  $\langle C, (g_n)_{n \in \mathbb{N}} \rangle$  is another cone, then the mediating map from  $C$  to  $D$  is given by  $g(x) = \langle g_n(x) : n \in \mathbb{N} \rangle$  or  $g = \bigsqcup_{n \in \mathbb{N}}^\uparrow e_n \circ g_n$ .
3.  $\langle D, (e_n)_{n \in \mathbb{N}} \rangle$  is a colimit of the diagram  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn})_{n \leq m \in \mathbb{N}} \rangle$ . If  $\langle E, (f_n)_{n \in \mathbb{N}} \rangle$  is another cocone, then the mediating map from  $D$  to  $E$  is given by  $f(\langle x_n : n \in \mathbb{N} \rangle) = \bigsqcup_{n \in \mathbb{N}}^\uparrow f_n(x_n)$  or  $f = \bigsqcup_{n \in \mathbb{N}}^\uparrow f_n \circ p_n$ .

**Proof.** We have already shown in Theorem 3.3.1 that a limit of the diagram  $\langle (D_n), (p_{nm}) \rangle$  is given by  $\langle D, (p_n) \rangle$  and that the mediating morphism has the (first) postulated form.

For the rest, let us start by showing that the functions  $e_m$  are well-defined, i.e. that  $y = e_m(x)$  is a commuting tuple. Assume  $n \leq n'$ . Then we have  $p_{nn'}(y_{n'}) = p_{nn'}(\bigsqcup_{k \sqsubseteq n', m}^\uparrow p_{nk} \circ e_{km}(x)) = \bigsqcup_{k \sqsubseteq n', m}^\uparrow p_{nn'} \circ p_{nk} \circ e_{km}(x) = \bigsqcup_{k \sqsubseteq n', m}^\uparrow p_{nk} \circ e_{km}(x) = y_n$ . The assignment  $x \mapsto e_m(x)$  is Scott-continuous because of general associativity and because only Scott-continuous maps are involved in the definition.

Next, let us now check that  $e_m$  and  $p_m$  form an e-p-pair.

$$\begin{aligned} e_m \circ p_m(\langle x_n : n \in \mathbb{N} \rangle) &= e_m(x_m) \\ &= \langle \bigsqcup_{k \sqsubseteq n, m}^\uparrow p_{nk} \circ e_{km}(x_m) : n \in \mathbb{N} \rangle \\ &= \langle \bigsqcup_{k \sqsubseteq n, m}^\uparrow p_{nk} \circ e_{km} \circ p_{mk}(x_k) : n \in \mathbb{N} \rangle \\ &\sqsubseteq \langle \bigsqcup_{k \sqsubseteq n, m}^\uparrow p_{nk}(x_k) : n \in \mathbb{N} \rangle \\ &= \langle x_n : n \in \mathbb{N} \rangle \end{aligned}$$

and  $p_m \circ e_m(x) = p_m(\langle \bigsqcup_{k \sqsubseteq n, m}^\uparrow p_{nk} \circ e_{km}(x) : n \in \mathbb{N} \rangle) = \bigsqcup_{k \sqsubseteq m}^\uparrow p_{mk} \circ e_{km}(x) = x$ .

A closer analysis reveals that  $e_m \circ p_m$  will leave all those elements of the tuple  $\langle x_n : n \in \mathbb{N} \rangle$  unchanged for which  $n \leq m$ :

$$p_n(e_m \circ p_m(\langle x_n : n \in \mathbb{N} \rangle)) = \dots = \bigsqcup_{k \geq n, m}^\uparrow p_{nk} \circ e_{km} \circ p_{mk}(x_k)$$

$$\begin{aligned}
&= \bigsqcup_{k \geq n, m}^{\uparrow} p_{nm} \circ p_{mk} \circ e_{km} \circ p_{mk}(x_k) \\
&= \bigsqcup_{k \geq n, m}^{\uparrow} p_{nm} \circ p_{mk}(x_k) = \bigsqcup_{k \geq n, m}^{\uparrow} x_n = x_n.
\end{aligned}$$

This proves that the  $e_m \circ p_m$ ,  $m \in \mathbb{N}$ , add up to the identity, as stated in (1). Putting this to use, we easily get the second representation for the mediating map into  $D$  viewed as a limit:  $g = \text{id} \circ g = \bigsqcup_{m \in \mathbb{N}}^{\uparrow} e_m \circ p_m \circ g = \bigsqcup_{m \in \mathbb{N}}^{\uparrow} e_m \circ g_m$ .

It remains to prove the universal property of  $D$  as a colimit. To this end let  $\langle E, (f_n)_{n \in \mathbb{N}} \rangle$  be a cocone over the expanding sequence. We have to check that  $f = \bigsqcup_{n \in \mathbb{N}}^{\uparrow} f_n \circ p_n$  is well-defined in the sense that the supremum is over a directed set. So let  $n \leq m$ . We get  $f_n \circ p_n = f_m \circ e_{mn} \circ p_{nm} \circ p_m \sqsubseteq f_m \circ p_m$ . It commutes with the colimiting maps because

$$\begin{aligned}
f \circ e_m &= \bigsqcup_{n \geq m}^{\uparrow} f_n \circ p_n \circ e_m \\
&= \bigsqcup_{n \geq m}^{\uparrow} f_n \circ p_n \circ e_n \circ e_{nm} \\
&= \bigsqcup_{n \geq m}^{\uparrow} f_n \circ e_{nm} = \bigsqcup_{n \geq m}^{\uparrow} f_m = f_m.
\end{aligned}$$

We also have to show that there is no other choice for  $f$ . Again the equation in (1) comes in handy: Let  $f'$  be any mediating morphism. It must satisfy  $f' \circ e_m = f_m$  and so  $f' \circ e_m \circ p_m = f_m \circ p_m$ . Forming the supremum on both sides gives  $f' = \bigsqcup_{m \in \mathbb{N}}^{\uparrow} f_m \circ p_m$  which is the definition of  $f$ . ■

This fact, that the colimit of an expanding sequence is canonically isomorphic to the limit of the associated dual diagram, is called the *limit-colimit coincidence*. It is one of the fundamental tools of domain theory and plays its most prominent role in the solution of recursive domain equations, Chapter 5. Because of this coincidence we will henceforth also speak of the *bilimit* of an expanding sequence and denote it by  $\text{bilim}(\langle D_n \rangle, (e_{mn}))$ .

We can generalize Theorem 3.3.7 in two ways; we can replace  $\mathbb{N}$  by an arbitrary directed set (in which case we will speak of an *expanding system*) and we can use general Scott-continuous adjunctions instead of e-p-pairs. The first generalization is harmless and does not need any serious adjustments in the proofs. We will freely use it from now on. The second, on the other hand, is quite interesting. By the passage from embeddings to, no longer injective, lower adjoints, we allow domains not only to grow but also to shrink as we move on in the index set. Thus points, which at some stage looked different, may at a later stage be recognized to be the same. The interested reader will find an outline of the mathematical theory

of this in the exercises. For the main text, we must remind ourselves that this generalization has so far not found any application in semantics.

Part (1) of the preceding theorem gives a characterization of bilimits:

**Lemma 3.3.8.** *Let  $\langle E, (f_n)_{n \in \mathbb{N}} \rangle$  be a cocone for the expanding sequence  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn}: D_n \rightarrow D_m)_{n \leq m \in \mathbb{N}} \rangle$ . It is colimiting if and only if, firstly, there are Scott-continuous functions  $g_n: E \rightarrow D_n$  such that each  $(f_n, g_n)$  is an e-p-pair and, secondly,  $\bigsqcup_{n \in \mathbb{N}}^\uparrow f_n \circ g_n = \text{id}_E$  holds.*

**Proof.** Necessity is Part (1) of Theorem 3.3.7. For sufficiency we show that the bilimit  $D$ , as constructed there, is isomorphic to  $E$ . We already have maps  $f: D \rightarrow E$  and  $g: E \rightarrow D$  because  $D$  is the bilimit. These commute with the limiting and the colimiting morphisms, respectively. So let us check that they compose to identities:

$$\begin{aligned} f \circ g(x) &= f(\langle g_n(x) : n \in \mathbb{N} \rangle) \\ &= \bigsqcup_{n \in \mathbb{N}}^\uparrow f_n \circ g_n(x) \\ &= x \end{aligned}$$

and

$$\begin{aligned} g \circ f &= (\bigsqcup_{n \in \mathbb{N}}^\uparrow e_n \circ g_n) \circ (\bigsqcup_{m \in \mathbb{N}}^\uparrow f_m \circ p_m) \\ &= \bigsqcup_{n \in \mathbb{N}}^\uparrow e_n \circ g_n \circ f_n \circ p_n \\ &= \bigsqcup_{n \in \mathbb{N}}^\uparrow e_n \circ p_n = \text{id}_D. \end{aligned}$$

■

We note that in the proof we have used the condition  $\bigsqcup_{n \in \mathbb{N}}^\uparrow f_n \circ g_n = \text{id}_E$  only for the first calculation. Without it, we still get that  $f$  and  $g$  form an e-p-pair. Thus we have:

**Proposition 3.3.9.** *Let  $\langle E, (f_n)_{n \in \mathbb{N}} \rangle$  be a cocone over the expanding sequence  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn}: D_n \rightarrow D_m)_{n \leq m \in \mathbb{N}} \rangle$  where the  $f_n$  are embeddings. Then the bilimit of the sequence is a sub-domain of  $E$ .*

In other words:

**Corollary 3.3.10.** *The bilimit of an expanding sequence is also the colimit (limit) in the restricted category of dcpos with embeddings (projections) as morphisms.*

### 3.3.3 Bilimits of domains

**Theorem 3.3.11.** *Let  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn}: D_n \rightarrow D_m)_{n \leq m \in \mathbb{N}} \rangle$  be an expanding sequence and  $\langle D, (e_n)_{n \in \mathbb{N}} \rangle$  its bilimit.*



1. If all  $D_n$  are  $(\omega)$ -continuous then so is  $D$ . If we are given bases  $B^n, n \in \mathbb{N}$  for each  $D_n$  then a basis for  $D$  is given by  $\bigcup_{n \in \mathbb{N}} e_n(B^n)$ .
2. If all  $D_n$  are  $(\omega)$ -algebraic then so is  $D$  and  $K(D) = \bigcup_{n \in \mathbb{N}} e_n(K(D_n))$ .

**Proof.** Given an element  $x \in D$  we first show that  $\bigcup_{n \in \mathbb{N}} e_n(B^n_{p_n(x)})$  is directed. To this end it is sufficient to show that for all  $n \leq m \in \mathbb{N}$  and for each  $y \in B^n_{p_n(x)}$  there is  $y' \in B^m_{p_m(x)}$  with  $e_n(y) \sqsubseteq e_m(y')$ . Well, because  $y$  approximates  $p_n(x)$  and because embeddings preserve the order of approximation, we have  $e_{mn}(y) \ll e_{mn}(p_n(x)) = e_{mn}(p_{mn} \circ p_m(x)) \sqsubseteq p_m(x)$ . Since  $p_m(x) = \bigsqcup^\uparrow B^m_{p_m(x)}$ , some  $y' \ll p_m(x)$  is above  $e_{mn}(y)$ . This implies  $e_n(y) = e_m(e_{mn}(y)) \sqsubseteq e_m(y')$ .

The set  $\bigcup_{n \in \mathbb{N}} e_n(B^n_{p_n(x)})$  gives back  $x$  because  $x = \bigsqcup^\uparrow_{n \in \mathbb{N}} e_n \circ p_n(x) = \bigsqcup^\uparrow_{n \in \mathbb{N}} e_n(\bigsqcup^\uparrow B^n_{p_n(x)}) = \bigsqcup^\uparrow_{n \in \mathbb{N}} \bigsqcup^\uparrow e_n(B^n_{p_n(x)}) = \bigsqcup^\uparrow \bigcup e_n(B^n_{p_n(x)})$ . It consists solely of approximants to  $x$  because the  $e_n$  are lower adjoints. ■

### Exercises 3.3.12.

1. Let  $D$  be a continuous domain and let  $f: D \rightarrow D$  be an idempotent Scott-continuous function. Show that  $f(x) \ll f(y)$  holds in the image of  $f$  if and only if there exists  $z \ll f(y)$  in  $D$  such that  $f(x) \sqsubseteq f(z) \sqsubseteq f(y)$ . In the case that  $D$  is algebraic, conclude that an element  $x$  of  $\text{im}(f)$  is compact if and only if there is  $c \in K(D)_{f(x)}$  with  $f(c) = f(x)$ .
2. Let  $p$  be a kernel operator with finite image. Show that  $\text{im}(p)$  is contained in  $K(D)$  and that  $p$  itself is compact in  $[D \rightarrow D]$ .
3. [Huth, 1992] A chain  $C$  is called *order dense* if for each pair  $x \sqsubset y$  there exists  $z \in C$  such that  $x \sqsubset z \sqsubset y$ .
  - (a) Let  $C$  be an order dense chain in an algebraic domain  $D$ . Construct a continuous idempotent function  $f$  on  $D$  with  $\text{im}(f) \sqsubseteq C$  and  $\text{im}(f)$  not algebraic (it must be continuous by Theorem 3.1.4).
  - (b) Let, conversely,  $f$  be a continuous and idempotent function on an algebraic dcpo  $D$  such that its image is not algebraic. Show that  $K(D)$  contains an order dense chain.
  - (c) An algebraic domain is called *retraction stable* if every idempotent on  $D$  has an algebraic image. Prove that an algebraic domain is retraction stable if and only if  $K(D)$  does not contain an order dense chain.
  - (d) Formulate a similar result for projection stable domains.
4. Let  $e: D \hookrightarrow E: p$  be an embedding projection pair between  $\sqcap$ -semi-lattices. Show that  $\text{im}(e)$  is a lower set in  $E$  if and only if for all  $x \sqsubseteq y$  in  $E$  we have  $e(p(x)) = e(p(y)) \sqcap x$ .
5. Formulate and prove a generalization of Proposition 3.1.13 for arbitrary posets.

6. Formulate an analogue of Proposition 3.2.4 for infinite products. Proceed as follows: First restrict to pointed dcpos. Next find an example of a (non-pointed) finite poset which has a non-algebraic infinite power. This should give you enough intuition to try the general case.
7. A dcpo may be seen as a topological space with respect to the Scott-topology. Given two dcpos we can form their product in **DCPO**. Show that the Scott-topology on the product need not be the product topology but that the two topologies coincide if one of the factors is a continuous domain.
8. Construct an example which shows that Lemma 3.2.6 does not hold for infinite products.
9. Derive Curry and composition as maps in an arbitrary cartesian closed category.
10. Let **C** be a cartesian closed full subcategory of **DCPO**. Let **R-C** be the full subcategory of **DCPO** whose objects are the retracts of objects of **C**. Show that **R-C** is cartesian closed.
11. Let  $\mathbb{Z}^-$  be the negative integers with the usual ordering. Show that the order of approximation on  $[\mathbb{Z}^- \rightarrow \mathbb{Z}^-]$  is empty. Find a pointed algebraic dcpo in which a similar effect takes place.
12. Show that **DCPO**<sub>⊥</sub> does not have coproducts.
13. Show that **CONT** does not have equalizers for all pairs of morphisms. (Hint: First convince yourself that limits in **CONT**, if they exist, have the same underlying dcpo as when they are calculated in **DCPO**.)
14. Complement the table in Section 3.2.6 with the infinitary counterparts of cartesian product, disjoint union, smash product and sum. Observe that for these the cardinality of the basis does play a role, so you have to add columns for  $\omega$ -**CONT** etc.
15. Show that the embeddings into the bilimit of an expanding sequence are given more concretely by  $e_m(x) = \langle x_n : n \in \mathbb{N} \rangle$  with

$$x_n = \begin{cases} p_{nm}(x), & n < m; \\ e_{nm}(x), & n \geq m. \end{cases}$$

Find a similar description for expanding systems.

16. Redo Section 3.3.2 for directed index sets and Scott-continuous adjunctions. The following are the interesting points:
  - (a) The limit-colimit coincidence, Theorem 3.3.7, holds verbatim.
  - (b) The characterization of bilimits given in Lemma 3.3.8 does not suffice. It states that  $E$  must not contain superfluous elements. Now we also need to say that  $E$  does not identify too many elements.

- (c) Given an expanding system  $\langle (D_i), (l_{ji}) \rangle$  with adjunctions, we can pass to quotient domains  $D'_i$  by setting  $D'_i = \text{im}(\bigsqcup_{k \sqsupseteq i}^\top u_{ik} \circ l_{ki})$ . Show that the original adjunctions when restricted and corestricted to the  $D'_i$  become e-p-pairs and that these define the same bilimit.
17. Let  $RD$  be the space of Scott-continuous idempotents on a dcpo  $D$ . Apply the previous exercise to show that  $\bigsqcup_{i \in I}^\top r_i = r$  in  $RD$  implies  $\text{bilim}(\text{im}(r_i)) \cong \text{im}(r)$  (where the connecting adjunctions are given by restricting the retractions to the respective image).
18. Prove that the Scott-topology on a bilimit of continuous domains is the restriction of the product topology on the product of the individual domains.

## 4 Cartesian closed categories of domains

In the last chapter we have seen that our big ambient categories **DCPO** and **DCPO** $_{\perp}$  are, among other things, cartesian closed and we have already pointed out that for the natural classes of *domains*, **CONT** and **ALG**, this is no longer true. The problematic construction is the exponential, which as we know by Lemma 3.2.7, must be the set of Scott-continuous functions ordered pointwise. If, on the other hand, we find a full subcategory of **CONT** which is closed under terminal object, cartesian product and function space, then it is also cartesian closed, because the necessary universal properties are inherited from **DCPO**.

Let us study more closely why function spaces may fail to be domains. The fact that the order of approximation may be empty tells us that there may be no natural candidates for basis elements in a function space. This we can better somewhat by requiring the image domain to contain a bottom element.

**Definition 4.0.1.** For  $D$  and  $E$  dcpos where  $E$  has a least element and  $d \in D, e \in E$ , we define the *step function*  $(d \searrow e): D \rightarrow E$  by

$$(d \searrow e)(x) = \begin{cases} e, & \text{if } d \ll x; \\ \perp_E, & \text{otherwise.} \end{cases}$$

More generally, we will use  $(O \searrow e)$  for the function which maps the Scott-open set  $O$  to  $e$  and everything else to  $\perp$ .

**Proposition 4.0.2.**

1. *Step functions are Scott-continuous.*
2. *Let  $D$  and  $E$  be dcpos where  $E$  is pointed and let  $f: D \rightarrow E$  be continuous. If  $e$  approximates  $f(d)$  then  $(d \searrow e)$  approximates  $f$ .*
3. *If, in addition,  $D$  and  $E$  are continuous then  $f$  is a supremum of step functions.*

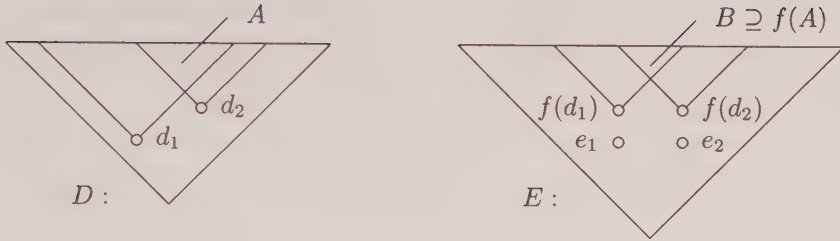


Fig. 10. Finding an upper bound for two step functions.

**Proof.** (1) Continuity follows from the openness of  $\uparrow d$ , respectively  $O$ .

(2) Let  $G$  be a directed family of functions with  $\bigsqcup^{\uparrow} G \sqsupseteq f$ . Suprema in  $[D \rightarrow E]$  are calculated pointwise so we also have  $\bigsqcup_{g \in G} g(d) \sqsupseteq f(d)$ . This implies that for some  $g \in G$ ,  $g(d) \sqsupseteq e$  holds. A simple case distinction then shows that  $g$  must be above  $(d \searrow e)$  everywhere.

(3) We show that for each  $d \in D$  and each  $e \ll f(d)$  in  $E$  there is a step function approximating  $f$  which maps  $d$  to  $e$ . Indeed, from  $d = \bigsqcup^{\uparrow} \downarrow d$  we get  $f(d) = f(\bigsqcup^{\uparrow}_{y \ll d} y) = \bigsqcup^{\uparrow}_{y \ll d} f(y)$  and so for some  $y \ll d$  we have  $f(y) \sqsupseteq e$ . The desired step function is therefore given by  $(y \searrow e)$ . Continuity of  $E$  implies that we can get arbitrarily close to  $f(d)$  this way. ■

Note that the supremum in (3) need not be directed, so we have *not* shown that  $[D \rightarrow E]$  is again continuous. Was it a mistake to require directedness for the set of approximants? The answer is no, because without it we could not have proved (3) in the first place.

The problem of joining finitely many step functions together, so as to build *directed* collections of approximants, comes up already in the case of two step functions  $(d_1 \searrow e_1)$  and  $(d_2 \searrow e_2)$  which approximate a given continuous function  $f$ . The situation is illustrated in Figure 10. The problem is where to map the (Scott-open but otherwise unstructured) set  $A = \uparrow d_1 \cap \uparrow d_2$ . It has to be done in such a way that the resulting function still approximates  $f$ . As it will turn out, it suffices to make special assumptions about *either* the image domain  $E$ —the topic of Section 4.1—*or* about the pre-image domain  $D$ —the topic of Section 4.2. In both cases we restrict our attention to pointed domains, and we work with step functions and joins of these. From these we can pass to more general domains, again in two ways. This will be outlined briefly in Section 4.3.2. The question then arises whether we have not missed out on some alternative way of building a cartesian closed category. This is not the case as we will see in Section 4.3. The basic tool for this fundamental result, Lemma 4.3.1, will nicely connect up with the dichotomy distinguishing Sections 4.1 and 4.2.



#### 4.1 Local uniqueness: Lattice-like domains

The idea for adjusting the image domain is simple; we assume that  $e_1$  and  $e_2$  have a least upper bound  $e$  (if bounded at all). Mapping the intersection  $A$  to  $e$  (and  $\uparrow d_1 \setminus A$  to  $e_1$  and  $\uparrow d_2 \setminus A$  to  $e_2$ ) results in a continuous function  $h$  which is above  $(d_1 \searrow e_1)$  and  $(d_2 \searrow e_2)$  and still approximates  $f$ . This is seen as follows: Suppose  $G$  is a directed collection of functions with supremum above  $f$ . Some  $g_1 \in G$  must be above  $(d_1 \searrow e_1)$  and some  $g_2 \in G$  must be above  $(d_2 \searrow e_2)$ . Then by construction every upper bound of  $\{g_1, g_2\}$  in  $G$  is above  $h$ .

In fact, we do not need that the join of  $e_1$  and  $e_2$  exists globally in  $E$ . It suffices to form the join for every  $a \in A$  inside  $\downarrow f(a)$ , because we have seen in Proposition 2.2.17 that all considerations about the order of approximation can be performed inside principal ideals. We have the following list of definitions.

**Definition 4.1.1.** Let  $E$  be a pointed continuous domain. We say that  $E$  is

1. an *L-domain*, if each pair  $e_1, e_2 \in E$  bounded by  $e \in E$  has a supremum in  $\downarrow e$ ;
2. a *bounded-complete domain* (or *bc-domain*), if each bounded pair  $e_1, e_2 \in E$  has a supremum;
3. (repeated for comparison) a *continuous lattice*, if each pair  $e_1, e_2 \in E$  has a supremum.

We denote the full subcategories of  $\mathbf{CONT}_\perp$  corresponding to these definitions by  $\mathbf{L}$ ,  $\mathbf{BC}$ , and  $\mathbf{LAT}$ . For the algebraic counterparts we use  $\mathbf{aL}$ ,  $\mathbf{aBC}$ , and  $\mathbf{aLAT}$ .

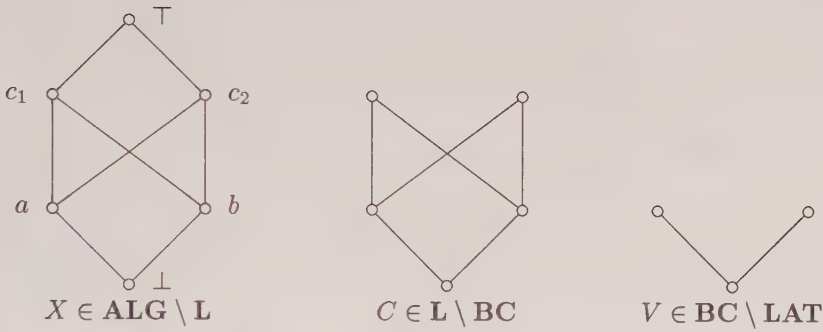
All this still makes sense if we forget about approximation but, surely, at this point the reader does not suffer from a lack of variety as far as categories are concerned. We would like to point out that continuous lattices are the main objects of study in [Gierz *et al.*, 1980], a mathematically oriented text, whereas the objects of  $\omega\text{-aBC}$  are often the domains of choice in semantics, where they appear under the name *Scott-domain*. Typical examples are depicted in Figure 11. They even characterize the corresponding categories, see Exercise 4.3.11(3).

Since domains have directed joins anyway, we see that in  $\mathbf{L}$ -domains every subset of a principal ideal has a supremum in that ideal. We also know that complete lattices can alternatively be characterized by infima. The same game can be played for the other two definitions:

**Proposition 4.1.2.** *Let  $D$  be a pointed continuous domain. Then  $D$  is an  $\mathbf{L}$ -domain, a  $\mathbf{bc}$ -domain, or a continuous lattice if and only if it has infima for bounded non-empty, non-empty, or arbitrary subsets, respectively.*

The consideration of infima may seem a side issue in the light of the





**Fig. 11.** Separating examples for the categories of lattice-like domains.

problem of turning function spaces into domains. Its relevance becomes clear when we remember that upper adjoints preserve infima. The second half of the following is therefore a simple observation. The first half follows from Proposition 3.1.2 and Theorem 3.1.4.

**Proposition 4.1.3.** *Retracts and bilimits of  $L$ -domains (bc-domains, continuous lattices) are again  $L$ -domains (bc-domains, continuous lattices).*

We can treat continuous and algebraic lattice-like domains nicely in parallel because the ideal completion respects these definitions:

**Proposition 4.1.4.** *Let  $D$  be an  $L$ -domain (bc-domain, continuous lattice). Then  $\text{Idl}(D, \sqsubseteq)$  is an algebraic  $L$ -domain (bc-domain, lattice).*

Thus  $\mathbf{L}$ ,  $\mathbf{BC}$ , and  $\mathbf{LAT}$  contain precisely the retracts of objects of  $\mathbf{aL}$ ,  $\mathbf{aBC}$ , and  $\mathbf{aLAT}$ , respectively. We conclude this section by stating the desired closure property of lattice-like domains.

**Proposition 4.1.5.** *Let  $D$  be a continuous domain and  $E$  an  $L$ -domain (bc-domain, continuous lattice). Then  $[D \rightarrow E]$  is again an  $L$ -domain (bc-domain, continuous lattice).*

**Corollary 4.1.6.** *The categories  $\mathbf{L}$ ,  $\mathbf{BC}$ ,  $\mathbf{LAT}$ , and their algebraic counterparts are cartesian closed.*

## 4.2 Finite choice: Compact domains

Let us now turn our attention to the first argument of the function space construction, which means by the general considerations from the beginning of this chapter, the study of open sets and their finite intersections. Step functions are defined using basic open sets of the form  $\uparrow d$ , and the fact that there is a single generator  $d$  was crucial in the proof that  $(d \searrow e)$  approximates  $f$  whenever  $e$  approximates  $f(d)$ . Arbitrary open sets are

unions of such basic opens (Proposition 2.3.6) but in general this is an infinite union and so the proof of Proposition 4.0.2 will no longer work. For the first time we have now reached a point in our exposition where the theory of algebraic domains is definitely simpler and better understood than that of continuous domains. Let us therefore treat this case first.

#### 4.2.1 Bifinite domains

Step functions ( $d \searrow e$ ) may in the algebraic case be defined using compact elements only, where the characteristic pre-image  $\uparrow d$  is actually equal to  $\uparrow d$ . Taking up our line of thought from above, we want for the algebraicity of the function space that the intersection  $A = \uparrow d_1 \cap \uparrow d_2$  is itself generated by finitely many compact points:  $A = \uparrow c_1 \cup \dots \cup \uparrow c_n$ . Note that the  $c_i$  must be minimal upper bounds of  $\{d_1, d_2\}$ . For each  $c_i$  we choose a compact element below  $f(c_i)$  and above  $e_1, e_2$ . New intersections then come up, this time between the different  $\uparrow c_i$ s. Let us therefore further assume that after finitely many iterations this process stops. It is an easy exercise to show that the function constructed in this way is a compact element below  $f$  and above  $(d_1 \searrow e_1)$  and  $(d_2 \searrow e_2)$ . We hope that this provides sufficient motivation for the following list of definitions.

**Definition 4.2.1.** Let  $P$  be a poset. (Think of  $P$  as the basis of an algebraic domain.)

1. We say that  $P$  is *mub-complete* (or: has *property m*) if for every upper bound  $x$  of a finite subset  $M$  of  $P$  there is a minimal upper bound of  $M$  below  $x$ . Written as a formula:  $\forall M \subseteq_{\text{fin}} P. \bigcap_{m \in M} \uparrow m = \uparrow \text{mub}(M)$ .
2. For a subset  $A$  of  $P$  let its *mub-closure*  $\text{mc}(A)$  be the smallest superset of  $A$  which for every finite  $M \subseteq \text{mc}(A)$  also contains  $\text{mub}(M)$ .
3. We say that  $P$  has the *finite mub property* if it is mub-complete and if every finite subset has a finite mub-closure. If, in addition,  $P$  has a least element, then we call it a *Plotkin-order*.
4. An algebraic domain whose basis of compact elements is a Plotkin-order is called a *bifinite domain*. The full subcategory of  $\mathbf{ALG}_\perp$  of bifinite domains we denote by  $\mathbf{B}$ .

With this terminology we can formulate precisely how finitely many step functions combine to determine a compact element in the function space [Abramsky, 1991b].

**Definition 4.2.2.** Let  $D$  be a bifinite domain and let  $E$  be pointed and algebraic. A finite subset  $F$  of  $K(D) \times K(E)$  is called *joinable* if

$$\forall G \subseteq F \exists H \subseteq F. (\pi_1(H) = \text{mub}(\pi_1(G)) \wedge \forall c \in \pi_2(G), d \in \pi_2(H). c \sqsubseteq d).$$

The function which we associate with a joinable family  $F$  is

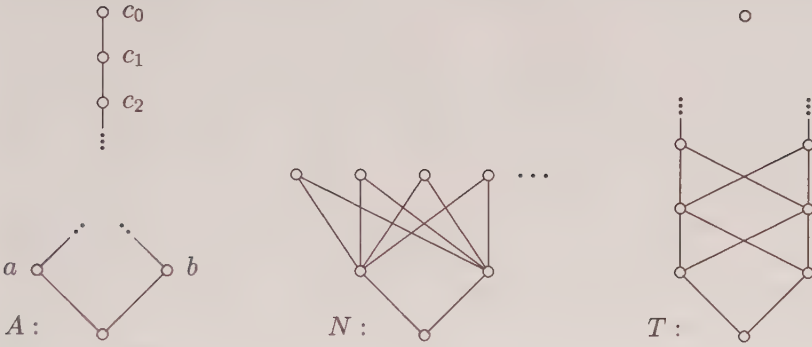


Fig. 12. Typical non-bifinite domains.

$$x \mapsto \bigsqcup \{e \mid \exists d \in K(D). d \sqsubseteq x \wedge (d, e) \in F\}.$$

**Lemma 4.2.3.** *If  $D$  is a bifinite domain and  $E$  is pointed and algebraic, then every joinable subset of  $K(D) \times K(E)$  gives rise to a compact element of  $[D \rightarrow E]$ .*

*If  $F$  and  $G$  are joinable families then the corresponding functions are related if and only if*

$$\forall (d, e) \in G \exists (d', e') \in F. d' \sqsubseteq d \text{ and } e \sqsubseteq e'.$$

The expected result, dual to Proposition 4.1.5 above, then is:

**Proposition 4.2.4.** *If  $D$  is a bifinite domain and  $E$  is pointed and algebraic, then  $[D \rightarrow E]$  is algebraic. All compact elements of  $[D \rightarrow E]$  arise from joinable families.*

Note that this is strictly weaker than Proposition 4.1.5 and we do not immediately get that  $\mathbf{B}$  is cartesian closed. For this we have to find alternative descriptions. The fact that we can get an algebraic function space by making special assumptions about *either* the argument domain *or* the target domain was noted in a very restricted form in [Markowsky, 1981].

The concept of finite mub closure is best explained by illustrating what can go wrong. In Figure 12 we have the three classical examples of algebraic domains which are not bifinite; in the first one the basis is not mub-complete, in the second one there is an infinite mub-set for two compact elements, and in the third one, although all mub-sets are finite, there occurs an infinite mub-closure. On a more positive note, it is clear that every finite and pointed poset is a Plotkin-order and hence bifinite. This

trivial example contains the key to a true understanding of bifiniteness; we will now prove that bifinite domains are precisely the bilimits of finite pointed posets.

**Proposition 4.2.5.** *Let  $D$  be an algebraic domain with mub-complete basis  $K(D)$  and let  $A$  be a set of compact elements. Then there is a least kernel operator  $p_A$  on  $D$  which keeps  $A$  fixed. It is given by  $p_A(x) = \biguparrow \{c \in \text{mc}(A) \mid c \sqsubseteq x\}$ .*

**Proof.** First note that  $p_A$  is well-defined because the supremum is indeed over a directed set. This follows from mub-completeness. Continuity follows from Corollary 2.2.16. On the other hand, it is clear that a kernel operator which fixes  $A$  must also fix each element of the mub-closure  $\text{mc}(A)$ , and so  $p_A$  is clearly the least monotone function with the desired property. ■

In a bifinite domain finite sets of compact elements have finite mub-closures. By the preceding proposition this implies that there are many kernel operators on such a domain which have a finite image. In fact, we get a directed family of them, because the order on kernel operators is completely determined by their images, Proposition 3.1.17. For the sake of brevity, let us call a kernel operator with finite image an *idempotent deflation*.

**Theorem 4.2.6.** *Let  $D$  be a pointed dcpo  $D$ . The following are equivalent:*

1.  $D$  is a bifinite domain.
2. There exists a directed collection  $(f_i)_{i \in I}$  of idempotent deflations of  $D$  whose supremum equals  $\text{id}_D$ .
3. The set of all idempotent deflations is directed and yields  $\text{id}_D$  as its join.

**Proof.** What we have not yet said is how algebraicity of  $D$  follows from the existence of idempotent deflations. For this observe that the inclusion of the image of a kernel operator is a lower adjoint and as such preserves compactness. For the implication ‘2  $\implies$  3’ we use the fact that idempotent deflations are in any case compact elements of the function space. ■

It is now only a little step to the promised categorical characterization.

**Theorem 4.2.7.** *A dcpo is bifinite if and only if it is a bilimit of an expanding system of finite pointed posets.*

**Proof.** Let  $D$  be bifinite and let  $(f_i)_{i \in I}$  be a family of idempotent deflations generating the identity. Construct an expanding system by taking as objects the images of the deflations and as connecting embeddings the inclusion of images. The associated upper adjoint is given by  $f_i$  restricted to  $\text{im}(f_j)$ .  $D$  is the bilimit of this system by Lemma 3.3.8.

If, conversely,  $\langle D, (f_i)_{i \in I} \rangle$  is a bilimit of finite posets then clearly the compositions  $f_i \circ g_i$ , where  $g_i$  is the upper adjoint of  $f_i$ , satisfy the requirements of Theorem 4.2.6. ■

So we have three characterizations of bifiniteness, the original one, which may be called an internal description, a functional description by Theorem 4.2.6, and a categorical one by Theorem 4.2.7. Often, the functional characterization is the most handy one in proofs. We should also mention that bifinite domains were first defined by Gordon Plotkin in [Plotkin, 1976] using expanding sequences. (In our taxonomy these are precisely the countably based bifinite domains.) The acronym he used for them, SFP, continues to be quite popular.

**Theorem 4.2.8.** *The category  $\mathbf{B}$  of bifinite domains is closed under cartesian product, function space, coalesced sum, and bilimits. In particular,  $\mathbf{B}$  is cartesian closed.*

**Proof.** Only function space and bilimit are non-trivial. We leave the latter as an exercise. For the function space let  $D$  and  $E$  be bifinite with families of idempotent deflations  $(f_i)_{i \in I}$  and  $(g_j)_{j \in J}$ . A directed family of idempotent deflations on  $[D \rightarrow E]$  is given by the maps  $F_{ij}: h \mapsto g_j \circ h \circ f_i$ ,  $\langle i, j \rangle \in I \times J$ . ■

### 4.2.2 FS-domains

Let us now look at continuous domains. The reasoning about what the structure of  $D$  should be in order to ensure that  $[D \rightarrow E]$  is continuous is pretty much the same as for algebraic domains. But at the point where we there introduced the mub-closure of a finite set of compact elements, we must now postulate the *existence* of some finite and finitely supported partitioning of  $D$ . This is clearly an increase in the logical complexity of our definition and also of doubtful practical use. It is more satisfactory to generalize the functional characterization.

**Definition 4.2.9.** Let  $D$  be a dcpo and  $f: D \rightarrow D$  be a Scott-continuous function. We say that  $f$  is *finitely separated* from the identity on  $D$ , if there exists a finite set  $M$  such that for any  $x \in D$  there is  $m \in M$  with  $f(x) \sqsubseteq m \sqsubseteq x$ . We speak of *strong separation* if for each  $x$  there are elements  $m, m' \in M$  with  $f(x) \sqsubseteq m \ll m' \sqsubseteq x$ .

A pointed dcpo  $D$  is called an *FS-domain* if there is a directed collection  $(f_i)_{i \in I}$  of continuous functions on  $D$ , each finitely separated from  $\text{id}_D$ , with the identity map as their supremum.

It is relatively easy to see that FS-domains are indeed continuous. Thus it makes sense to speak of **FS** as the full subcategory of **CONT** where the objects are the FS-domains.

We have exact parallels to the properties of bifinite domains, but often the proofs are trickier.



**Proposition 4.2.10.** *If  $D$  is an FS-domain and  $E$  is pointed and continuous, then  $[D \rightarrow E]$  is continuous.*

**Theorem 4.2.11.** *The category **FS** is closed under the formation of products, function spaces, coalesced sums, and bilimits. It is cartesian closed.*

What we do not have are a categorical characterization or a description of FS-domains as retracts of bifinite domains. All we can say is the following.

**Proposition 4.2.12.**

1. *Every bifinite domain is an FS-domain.*
2. *A retract of an FS-domain is an FS-domain.*
3. *An algebraic FS-domain is bifinite.*

To fully expose our ignorance, we conclude this subsection with an example of a well-structured FS-domain of which we do not know whether it is a retract of a bifinite domain.

**Example.** Let *Disc* be the collection of all closed discs in the plane plus the plane itself, ordered by reversed inclusion. One checks that the filtered intersection of discs is again a disc, so *Disc* is a pointed dcpo. A disc  $d_1$  approximates a disc  $d_2$  if and only if  $d_1$  is a neighborhood of  $d_2$ . This proves that *Disc* is continuous. For every  $\epsilon > 0$  we define a map  $f_\epsilon$  on *Disc* as follows. All discs inside the open disc with radius  $\frac{1}{\epsilon}$  are mapped to their closed  $\epsilon$ -neighborhood, all other discs are mapped to the plane which is the bottom element of *Disc*. Because the closed discs contained in some compact set form a compact space under the Hausdorff subspace topology, these functions are finitely separated from the identity map. This proves that *Disc* is a countably based FS-domain.

### 4.2.3 Coherence

This is a good opportunity to continue our exposition of the topological side of domain theory, which we began in Section 2.3. We need a second tool complementing the lattice  $\sigma_D$  of Scott-open sets, namely, the compact saturated sets. Here ‘compact’ is to be understood in the classical topological sense of the word, i.e. a set  $A$  of a topological space is *compact* if every covering of  $A$  by open sets contains a finite subcovering. *Saturated* are those sets which are intersections of their neighborhoods. In dcpos equipped with the Scott-topology these are precisely the upper sets, as is easily seen using opens of the form  $D \setminus \downarrow x$ .

**Theorem 4.2.13.** *Let  $D$  be a continuous domain. The sets of open neighborhoods of compact saturated sets are precisely the Scott-open filters in  $\sigma_D$ .*

By Proposition 7.2.27 this is a special case of the Hofmann–Mislove Theorem 7.2.9.

Let us denote the set of compact saturated sets of a dcpo  $D$ , ordered by

reverse inclusion, by  $\kappa_D$ . We will refer to families in  $\kappa_D$  which are directed with respect to reverse inclusion more concretely as filtered families. The following, then, is only a re-formulation of Corollary 7.2.11.

**Proposition 4.2.14.** *Let  $D$  be a continuous domain.*

1.  $\kappa_D$  is a dcpo. Directed suprema are given by intersection.
2. If the intersection of a filtered family of compact saturated sets is contained in a Scott-open set  $O$  then some element of it belongs to  $O$  already.
3.  $\kappa_D \setminus \{\emptyset\}$  is a dcpo.

**Proposition 4.2.15.** *Let  $D$  be a continuous domain.*

1.  $\kappa_D$  is a continuous domain.
2.  $A \ll B$  holds in  $\kappa_D$  if and only if there is a Scott-open set  $O$  with  $B \subseteq O \subseteq A$ .
3.  $O \ll U$  holds in  $\sigma_D$  if and only if there is a compact saturated set  $A$  with  $O \subseteq A \subseteq U$ .

**Proof.** All three claims are shown easily using upper sets generated by finitely many points: If  $O$  is an open neighborhood of a compact saturated set  $A$  then there exists a finite set  $M$  of points of  $O$  with  $A \subseteq \uparrow M \subseteq \uparrow M \subseteq O$ . ■

The interesting point about FS-domains then is that their space of compact saturated sets is actually a continuous lattice. We already have directed suprema (in the form of filtered intersections) and continuity, so this boils down to the property that the intersection of two compact saturated sets is again compact. Let us call domains for which this is true *coherent domains*. Given the intimate connection between  $\sigma_D$  and  $\kappa_D$ , it is no surprise that we can read off coherence from the lattice of open sets.

**Proposition 4.2.16.** *A continuous domain  $D$  is coherent if and only if for all  $O, U_1, U_2 \in \sigma_D$  with  $O \ll U_1$  and  $O \ll U_2$  we also have  $O \ll U_1 \cap U_2$ .*

(In Figure 6 we gave an example showing that the condition is not true in arbitrary continuous lattices.)

This result specializes for algebraic domains as follows:

**Proposition 4.2.17.** *An algebraic domain  $D$  is coherent if and only if  $K(D)$  is mub-complete and finite sets of  $K(D)$  have finite sets of minimal upper bounds.*

This proposition was named the ‘2/3-SFP Theorem’ in [Plotkin, 1981] because coherence rules out precisely the first two non-examples of Plotkin-orders, Figure 12, but not the third. The only topological characterization of bifinite domains we have at the moment makes use of the continuous function space, see Lemma 4.3.2.

We observe that for algebraic coherent domains,  $\sigma_D$  and  $\kappa_D$  have a common sublattice, namely that of compact-open sets. These are precisely the sets of the form  $\uparrow c_1 \cup \dots \cup \uparrow c_n$  with the  $c_i$  compact elements. This lattice generates both  $\sigma_D$  and  $\kappa_D$  when we form arbitrary suprema. This pleasant coincidence features prominently in Chapter 7.

**Theorem 4.2.18.** *FS-domains (bifinite domains) are coherent.*

Let us reformulate the idea of coherence in yet another way.

**Definition 4.2.19.** The *Lawson-topology* on a dcpo  $D$  is the smallest topology containing all Scott-open sets and all sets of the form  $D \setminus \uparrow x$ . It is denoted by  $\lambda_D$ .

**Proposition 4.2.20.** *Let  $D$  be a continuous domain.*

1. *The Lawson-topology on  $D$  is Hausdorff.*
2. *Every upper Lawson-open set is also Scott-open.*
3. *Every lower Lawson-open set is of the form  $D \setminus A$  where  $A$  is Scott-compact saturated.*
4. *The Lawson-topology on  $D$  is compact if and only if  $D$  is coherent.*
5. *A Scott-continuous retract of a Lawson-compact continuous domain is Lawson-compact and continuous.*

So we see that FS-domains and bifinite domains carry a natural compact Hausdorff topology. We will make use of this in Chapter 6.

### 4.3 The hierarchy of categories of domains

The purpose of this section is to show that there are no other ways of constructing a cartesian closed full subcategory of **CONT** or **ALG** than those exhibited in the previous two sections. The idea that such a result could hold originated with Gordon Plotkin, [Plotkin, 1981]. For the particular class  $\omega\text{-ALG}_\perp$  it was verified by Mike Smyth in [Smyth, 1983a], for the other classes by Achim Jung in [1988; 1989; 1990]. All these classification results depend on the axiom of choice.

#### 4.3.1 Domains with least element

Let us start right away with the crucial bifurcation lemma on which everything else in this section is based.

**Lemma 4.3.1.** *Let  $D$  and  $E$  be continuous domains, where  $E$  is pointed, such that  $[D \longrightarrow E]$  is continuous. Then  $D$  is coherent or  $E$  is an L-domain.*

**Proof.** By contradiction. Assume  $D$  is not coherent and  $E$  is not an L-domain. By Proposition 4.2.16 there exist open sets  $O, U_1$ , and  $U_2$  in  $D$  such that  $O \ll U_1$  and  $O \ll U_2$  hold but not  $O \ll U_1 \cap U_2$ . Therefore there is a directed collection  $(V_i)_{i \in I}$  of open sets covering  $U_1 \cap U_2$ , none of

which covers  $O$ . We shall also need interpolating sets  $U'_1$  and  $U'_2$ , that is,  $O \ll U'_1 \ll U_1$  and  $O \ll U'_2 \ll U_2$ .

The assumption about  $E$  not being an L-domain can be transformed into two special cases. Either  $E$  contains the algebraic domain  $A$  from Figure 12 (where the descending chain in  $A$  may generally be an ordinal) or  $X$  from Figure 11 as a retract. We have left the proof of this as Exercise 4.3.11(3). Note that if  $E'$  is a retract of  $E$  then  $[D \rightarrow E']$  is a retract of  $[D \rightarrow E]$  and hence the former is continuous if the latter is. Let us now prove for both cases that  $[D \rightarrow E]$  is not continuous.

Case 1:  $E = A$ . Consider the step functions  $f_1 = (U'_1 \searrow a)$  and  $f_2 = (U'_2 \searrow b)$ . They clearly approximate  $f$ , which is defined by

$$f(x) = \begin{cases} c_0, & \text{if } x \in U_1 \cap U_2; \\ a, & \text{if } x \in U_1 \setminus U_2; \\ b, & \text{if } x \in U_2 \setminus U_1; \\ \perp, & \text{otherwise.} \end{cases}$$

Since approximating sets are directed we ought to find an upper bound  $g$  for  $f_1$  and  $f_2$  approximating  $f$ . But this impossible: Given an upper bound of  $\{f_1, f_2\}$  below  $f$  we have the directed collection  $(h_i)_{i \in I}$  defined by

$$h_i(x) = \begin{cases} c_0, & \text{if } x \in V_i; \\ c_{n+1}, & \text{if } x \in (U_1 \cap U_2) \setminus V_i \text{ and } g(x) = c_n; \\ g(x), & \text{otherwise.} \end{cases}$$

No  $h_i$  is above  $g$  because  $(U_1 \cap U_2) \setminus V_i$  must contain a non-empty piece of  $O$  and there  $h_i$  is strictly below  $g_i$ . The supremum of the  $h_i$ , however, equals  $f$ . Contradiction.

Case 2:  $E = X$ . We choose open sets in  $D$  as in the previous case. The various functions, giving the contradiction, are now defined by  $f_1 = (U'_1 \searrow a)$ ,  $f_2 = (U'_2 \searrow b)$ ,

$$f(x) = \begin{cases} c_1, & \text{if } x \in U_1 \cap U_2; \\ a, & \text{if } x \in U_1 \setminus U_2; \\ b, & \text{if } x \in U_2 \setminus U_1; \\ \perp, & \text{otherwise.} \end{cases}$$

$$h_i(x) = \begin{cases} \top, & \text{if } x \in V_i; \\ c_2, & \text{if } x \in (U_1 \cap U_2) \setminus V_i; \\ g(x), & \text{otherwise.} \end{cases}$$

The remaining problem is that coherence does not imply that  $D$  is an FS-domain (nor, in the algebraic case, that it is bifinite). It is taken care of by passing to higher-order function spaces:

**Lemma 4.3.2.** *Let  $D$  be a continuous domain with bottom element. Then  $D$  is an FS-domain if and only if both  $D$  and  $[D \rightarrow D]$  are coherent.*

(The proof may be found in [Jung, 1990].)

Combining the preceding two lemmas with Lemmas 3.2.5 and 3.2.7 we get the promised classification result.

**Theorem 4.3.3.** *Every cartesian closed full subcategory of  $\mathbf{CONT}_\perp$  is contained in  $\mathbf{FS}$  or  $\mathbf{L}$ .*

Adding Proposition 4.2.12 we get the analogue for algebraic domains:

**Theorem 4.3.4.** *Every cartesian closed full subcategory of  $\mathbf{ALG}_\perp$  is contained in  $\mathbf{B}$  or  $\mathbf{aL}$ .*

Forming the function space of an L-domain may in general increase the cardinality of the basis (Exercise 4.3.11(17)). If we restrict the cardinality, this case is ruled out:

**Theorem 4.3.5.** *Every cartesian closed full subcategory of  $\omega\text{-}\mathbf{CONT}_\perp$  ( $\omega\text{-}\mathbf{ALG}_\perp$ ) is contained in  $\omega\text{-}\mathbf{FS}$  ( $\omega\text{-}\mathbf{B}$ ).*

### 4.3.2 Domains without least element

The classification of pointed domains, as we have just seen, is governed by the dichotomy between coherent and lattice-like structures. Expressed at the element level, and at least for algebraic domains we have given the necessary information, it is the distinction between finite mub-closures and locally unique suprema of finite sets. It turns out that passing to domains which do not necessarily have bottom elements implies that we also have to study the mub-closure of the empty set. We get again the same dichotomy. Coherence in this case means that  $D$  itself, that is, the largest element of  $\sigma_D$ , is a compact element. This is just the compactness of  $D$  as a topological space. And the property that  $E$  is lattice-like boils down to the requirement that each element of  $E$  is above a unique minimal element, so  $E$  is really the disjoint union of pointed components.

**Lemma 4.3.6.** *Let  $D$  and  $E$  be continuous domains such that  $[D \rightarrow E]$  is continuous. Then  $D$  is compact or  $E$  is a disjoint union of pointed domains.*

The proof is a cut-down version of that of Lemma 4.3.1 above. The surprising fact is that this choice can be made *independently* from the choice between coherent domains and L-domains. Before we state the classification, which because of this independence, will now involve  $2 \times 2 = 4$  cases, we have to refine the notion of compactness, because just like coherence it is not the full condition necessary for cartesian closure.

**Definition 4.3.7.** A dcpo  $D$  is a *finite amalgam* if it is the union of finitely many pointed dcpos  $D_1, \dots, D_n$  such that every intersection of  $D_i$ 's is also a union of  $D_i$ 's. (Compare the definition of mub-complete.)



For categories whose objects are finite amalgams of objects from another category **C** we use the notation **F-C**. Similarly, we write **U-C** if the objects are disjoint unions of objects of **C**.

**Proposition 4.3.8.** *A mub-complete dcpo is a finite amalgam if and only if the mub-closure of the empty set is finite.*

**Lemma 4.3.9.** *If both  $D$  and  $[D \rightarrow D]$  are compact and continuous then  $D$  is a finite amalgam.*

**Theorem 4.3.10.**

1. *The maximal cartesian closed full subcategories of **CONT** are **F-FS**, **U-FS**, **F-L**, and **U-L**.*
2. *The maximal cartesian closed full subcategories of **ALG** are **F-B**, **U-B**, **F-aL**, and **U-aL**.*

At this point we can answer a question that may have occurred to the diligent reader some time ago, namely, why we have defined bifinite domains in terms of *pointed* finite posets, where clearly we never needed the bottom element in the characterizations of them. The answer is that we wanted to emphasize the uniform way of passing from pointed to general domains. The fact that the objects of **F-B** can be represented as bilimits of finite posets is then just a pleasant coincidence.

**Exercises 4.3.11.**

1. [Jung, 1989] Show that a dcpo  $D$  is continuous if the function space  $[D \rightarrow D]$  is continuous.
2. Let  $D$  be a bounded-complete domain. Show that ' $\sqcap$ ' is a Scott-continuous function from  $D \times D$  to  $D$ .
3. Characterize the lattice-like (pointed) domains by forbidden substructures:
  - (a)  $E$  is  $\omega$ -continuous but not mub-complete if and only if domain  $A$  in Figure 12 is a retract of  $E$ .
  - (b)  $E$  is mub-complete but not an L-domain if and only if domain  $X$  in Figure 11 is a retract of  $E$ .
  - (c)  $E$  is an L-domain but not bounded-complete if and only if domain  $C$  in Figure 11 is a retract of  $E$ .
  - (d)  $E$  is a bounded-complete domain but not a lattice if and only if domain  $V$  in Figure 11 is a retract of  $E$ .
4. Find a poset in which all pairs have finite mub-closures but in which a triple of points exists with infinite mub-closure.
5. Show that if for an algebraic domain  $D$  the basis is mub-complete then  $D$  itself is not necessarily mub-complete.
6. Show that in a bifinite domain finite sets of non-compact elements may have infinitely many minimal upper bounds and, even if these

are all finite, may have infinite mub-closures.

7. Show that if  $A$  is a subset of an L-domain, then  $A \cup \text{mub}(A)$  is mub-closed.
8. Prove that bilimits of bifinite domains are bifinite.
9. Prove the following statements about retracts of bifinite domains.
  - (a) A pointed dcpo  $D$  is a retract of a bifinite domain if and only if there is a directed family  $(f_i)_{i \in I}$  of functions on  $D$  such that each  $f_i$  has a finite image and such that  $\bigcup_{i \in I}^\uparrow f_i = \text{id}_D$ . (You may want to do this for countably based domains first.)
  - (b) The ideal completion of a retract of a bifinite domain need not be bifinite.
  - (c) If  $D$  is a countably based retract of a bifinite domain then it is also the image of a projection from a bifinite domain. (Without countability this is an open problem.)
  - (d) The category of retracts of bifinite domains is cartesian closed and closed under bilimits.
10. Prove that FS-domains have infima for downward directed sets. As a consequence, an FS-domain which has binary infima is a bc-domain.
11. Show that in a continuous domain the Lawson-closed upper sets are precisely the Scott-compact saturated sets.
12. Characterize Lawson-continuous maps between bifinite domains.
13. We have seen that every bifinite domain is the bilimit of finite posets. As such, it can be thought of as a subset of the product of all these finite posets. Prove that the Lawson-topology on the bifinite domain is the restriction of the product topology if each finite poset is equipped with the discrete topology.
14. Prove that a coherent L-domain is an FS-domain.
15. Characterize those domains which are both L-domains and FS-domains.
16. Characterize Scott-topology and Lawson-topology on both L-domains and FS-domains by the ideal of functions approximating the identity.
17. [Jung, 1989] Let  $E$  be an L-domain such that  $[E \longrightarrow E]$  is countably based. Show that  $E$  is an FS-domain.

## 5 Recursive domain equations

The study of recursive domain equations is not easily motivated by reference to other mathematical structure theories. So we shall allow ourselves to deviate from our general philosophy and spend some time on examples. Beyond motivation, our examples represent three different (and almost disjoint) areas in which recursive domain equations arise, in which they serve a particular role, and in which particular aspects about solutions become

prominent. It is an astonishing fact that within domain theory all these aspects are dealt with in a unified and indeed very satisfactory manner. This richness and interconnectedness of the theory of recursive domain equations, beautiful as it is, may nevertheless appear quite confusing on a first encounter. As a general guideline we offer the following: recursive domain equations and the domain theory for solving them comprise a *technique* that is worth *learning*. But in order to *understand* the *meaning* of a particular recursive domain equation, you have to know the context in which it came up.

## 5.1 Examples

### 5.1.1 Genuine equations

The prime example here is  $X \cong [X \longrightarrow X]$ . Solving this equation in a cartesian closed category gives a model for the untyped  $\lambda$ -calculus [Scott, 1980; Barendregt, 1984], in which, as we know, no type distinction is made between functions and arguments. When setting up an interpretation of  $\lambda$ -terms with values in  $D$ , where  $D$  solves this equation, we need the isomorphisms  $\phi: D \rightarrow [D \longrightarrow D]$  and  $\psi: [D \longrightarrow D] \rightarrow D$  explicitly. We conclude that even in the case of a genuine equation we are looking not only for an object but an object *plus* an isomorphism. This is a first hint that we shall need to treat recursive domain equations in a categorical setting. However, the function space operator is contravariant in its first and covariant in its second argument and so there is definitely an obstacle to overcome. A second problem that this example illustrates is that there may be many solutions to choose from. How do we recognize a canonical one? This will be the topic of Section 5.3.

Besides this classical example, genuine equations are rare. They come up in semantics when one is confronted with the ability of computers to treat information both as program text and as data.

### 5.1.2 Recursive definitions

In semantics we sometimes need to make recursive definitions, for very much the same reasons that we need recursive function calls, namely, we sometimes do not know how often the body of a definition (resp. function) needs to be repeated. To give an example, take the following definition of a space of so-called ‘resumptions’:

$$R \cong [S \longrightarrow (S \oplus S \times R)].$$

We read it as follows: A resumption is a map which assigns to a state either a final state or an intermediary state together with another resumption representing the remaining computation. Such a recursive definition is therefore nothing but a shorthand for an infinite (but regular) expression. Likewise, a while loop could be replaced by an infinite repetition of its body.

This analogy suggests that the way to give meaning to a recursive definition is to seek a limit of the repeated unwinding of the body of the definition starting from a trivial domain. No doubt this is in accordance with our intuition, and indeed this is how we shall solve equations in general. But again, before we can do this, we need to be able to turn the right-hand side of the specification into a functor.

### 5.1.3 Data types

Data types are algebras, i.e. sets together with operations. The study of this notion is known as ‘algebraic specification’ [Ehrig and Mahr, 1985] or ‘initial algebra semantics’ [Goguen *et al.*, 1978]. We choose a formulation which fits nicely into our general framework.

**Definition 5.1.1.** Let  $F$  be a functor on a category  $\mathbf{C}$ . An  $F$ -algebra is given by an object  $A$  and a map  $f: F(A) \rightarrow A$ . A homomorphism between algebras  $f: F(A) \rightarrow A$  and  $f': F(A') \rightarrow A'$  is a map  $g: A \rightarrow A'$  such that the following diagram commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(g)} & F(A') \\ f \downarrow & & \downarrow f' \\ A & \xrightarrow{g} & A' \end{array}$$

For example, if we let  $F$  be the functor over **Set** which assigns  $\mathbb{I} \dot{\cup} \mathbb{A} \times \mathbb{A}$  to  $A$ , then  $F$ -algebras are precisely the algebras with one nullary and one binary operation in the sense of universal algebra. Lehmann and Smyth [Lehmann and Smyth, 1981] discuss many examples. Many of the data types which programming languages deal with are furthermore totally free algebras, or term algebras on no generators. These are distinguished by the fact that there is precisely one homomorphism from them into any other algebra of the same signature. In our categorical language we express this by initiality. Term algebras (alias initial  $F$ -algebras) are connected with the topic of this chapter because of the following observation:

**Lemma 5.1.2.** *If  $i: F(A) \rightarrow A$  is an initial  $F$ -algebra then  $i$  is an isomorphism.*

**Proof.** Consider the following composition of homomorphisms:



$$\begin{array}{ccccc}
 F(A) & \xrightarrow{F(f)} & F^2(A) & \xrightarrow{F(i)} & F(A) \\
 \downarrow i & & \downarrow F(i) & & \downarrow i \\
 A & \xrightarrow{f} & F(A) & \xrightarrow{i} & A
 \end{array}$$

where  $f$  is the unique homomorphism from  $i: F(A) \rightarrow A$  to  $F(i): F^2(A) \rightarrow F(A)$  guaranteed by initiality. Again by initiality,  $i \circ f$  must be  $\text{id}_A$ . And from the first quadrangle we get  $f \circ i = F(i) \circ F(f) = F(\text{id}_A) = \text{id}_{F(A)}$ . So  $f$  and  $i$  are inverses of each other. ■

So in order to find an initial  $F$ -algebra, we need to solve the equation  $X \cong F(X)$ . But once we get a solution, we still have to check initiality, that is, we must validate that the isomorphism from  $F(X)$  to  $X$  is the right structure map.

In category theory we habitually dualize all definitions. In this case we get (final) co-algebras. Luckily, this concept is equally meaningful. Where the map  $f: F(A) \rightarrow A$  describes how new objects of type  $A$  are *constructed* from old ones, a map  $g: A \rightarrow F(A)$  stands for the opposite process, the *decomposition* of an object into its constituents. Naturally, we want the two operations to be inverses of each other. In other words, if  $i: F(A) \rightarrow A$  is an initial  $F$ -algebra, then we require  $i^{-1}: A \rightarrow F(A)$  to be the final co-algebra.

Peter Freyd [1991] makes this reasoning the basis of an axiomatic treatment of domain theory. Beyond and above axiomatizing known results, he treats contravariant and mixed variant functors and offers a universal property encompassing both initiality and finality. This will allow us to judge the solution of general recursive domain equations with respect to canonicity.

## 5.2 Construction of solutions

Suppose we are given a recursive domain equation  $X \cong F(X)$  where the right-hand side defines a functor on a suitable category of domains. As suggested by the example in Section 5.1.2, we want to repeat the trick which gave us fixpoints for Scott-continuous functions, namely, to take a (bi-)limit of the sequence  $\mathbb{I}, \mathbb{F}(\mathbb{I}), \mathbb{F}(\mathbb{F}(\mathbb{I})), \dots$ . Remember that bilimits are defined in terms of e-p-pairs. This makes it necessary that we, at least temporarily, switch to a different category. The convention that we adopt for this chapter is to let  $\mathbf{D}$  stand for any category of *pointed* domains, closed under bilimits. All the cartesian closed categories of pointed domains mentioned in Chapter 4 qualify. We denote the corresponding subcategory where the morphisms are embeddings by  $\mathbf{D}^e$ . Some results will only hold for strict functions. Recall that our notation for these was  $f: D \xrightarrow{\perp!} E$  and



$\mathbf{D}_{\perp!}$  for categories. Despite this unhappy (but unavoidable) proliferation of categories, recall that the central limit–colimit Theorem 3.3.7 and Corollary 3.3.10 state a close connection: Colimits of expanding sequences in  $\mathbf{D}^e$  are also colimits in  $\mathbf{D}$  and, furthermore, if the embeddings defining the sequence are replaced by their upper adjoints, the colimit coincides with the corresponding limit. This will bear fruit when we analyse the solutions we get in  $\mathbf{D}^e$  from various angles as suggested by the examples in the last subsection.

Let us now start by just assuming that our functor restricts to  $\mathbf{D}^e$ .

### 5.2.1 Continuous functors

**Definition 5.2.1.** A functor  $F: \mathbf{D}^e \rightarrow \mathbf{D}^e$  is called *continuous* if for every expanding sequence  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn}: D_n \rightarrow D_m)_{n \sqsubseteq m \in \mathbb{N}} \rangle$  with colimit  $\langle D, (e_n)_{n \in \mathbb{N}} \rangle$ , we have that  $\langle F(D), (F(e_n))_{n \in \mathbb{N}} \rangle$  is a colimit of the sequence  $\langle (F(D_n))_{n \in \mathbb{N}}, (F(e_{mn}): F(D_n) \rightarrow F(D_m))_{n \sqsubseteq m \in \mathbb{N}} \rangle$ .

This, obviously, is Scott-continuity expressed for functors. Whether we formulate it in terms of expanding sequences or expanding systems is immaterial. The question is not what is allowed to enter the model, but rather, how much do I have to check before I can apply the theorems in this chapter. And sequences are all that is needed.

This, then, is the central lemma on which our domain theoretic technique for solving recursive domain equations is based (recall that  $f^*$  is our notation for the upper adjoint of  $f$ ):

**Lemma 5.2.2.** *Let  $F$  be a continuous functor on a category  $\mathbf{D}^e$  of domains. For each embedding  $e: A \rightarrow F(A)$  consider the colimit  $\langle D, (e_n)_{n \in \mathbb{N}} \rangle$  of the expanding sequence  $A \xrightarrow{e} F(A) \xrightarrow{F(e)} F(F(A)) \xrightarrow{F(F(e))} \dots$ . Then  $D$  is isomorphic to  $F(D)$  via the maps*

$$\begin{aligned} \text{fold} &= \bigsqcup_{n \in \mathbb{N}}^\uparrow e_{n+1} \circ F(e_n)^* &: F(D) \rightarrow D, \text{ and} \\ \text{unfold} &= \bigsqcup_{n \in \mathbb{N}}^\uparrow F(e_n) \circ e_{n+1}^* &: D \rightarrow F(D). \end{aligned}$$

For each  $n \in \mathbb{N}$  they satisfy the equations

$$\begin{aligned} F(e_n) &= \text{unfold} \circ e_{n+1} \\ F(e_n)^* &= e_{n+1}^* \circ \text{fold}. \end{aligned}$$

**Proof.** We know that  $\langle D, (e_n)_{n \in \mathbb{N} \setminus \{0\}} \rangle$  is a colimit over the diagram

$$F(A) \xrightarrow{F(e)} F(F(A)) \xrightarrow{F(F(e))} \dots$$

(clipping off the first approximation makes no difference), where there is also the cocone  $\langle F(D), (F(e_n))_{n \in \mathbb{N}} \rangle$ . The latter is also colimiting by the

continuity of  $F$ . In this situation Theorem 3.3.7 provides us with unique mediating morphisms which are precisely the stated fold and unfold. They are inverses of each other because both cocones are colimiting. The equations follow from the explicit description of mediating morphisms in Theorem 3.3.7. ■

Note that since we have restricted attention to pointed domains, we always have the initial embedding  $e: \mathbb{I} \rightarrow F(\mathbb{I})$ . The solution to  $X \cong F(X)$  based on this embedding we call *canonical* and denote it by  $\text{FIX}(F)$ .

### 5.2.2 Local continuity

Continuity of a functor is a hard condition to verify. Luckily there is a property which is stronger but nevertheless much easier to check. It will also prove useful in the next section.

**Definition 5.2.3.** A functor  $F$  from  $\mathbf{D}$  to  $\mathbf{E}$ , where  $\mathbf{D}$  and  $\mathbf{E}$  are categories of domains, is called *locally continuous*, if the maps  $\text{Hom}(D, D') \rightarrow \text{Hom}(F(D), F(D'))$ ,  $f \mapsto F(f)$ , are continuous for all objects  $D$  and  $D'$  from  $\mathbf{D}$ .

**Proposition 5.2.4.** A locally continuous functor  $F: \mathbf{D} \rightarrow \mathbf{E}$  restricts to a continuous functor from  $\mathbf{D}^e$  to  $\mathbf{E}^e$ .

We will soon generalize this, so there is no need for a proof at this point.

Typically, recursive domain equations are built from the basic constructions listed in Section 3.2. The strategy is to check local continuity for each of these individually and then rely on the fact that composition of continuous functors yields a continuous functor. However, we must realize that the function space construction is contravariant in its first and covariant in its second variable, and so the technique from the preceding paragraph does not immediately apply. Luckily, it can be strengthened to cover this case as well.

**Definition 5.2.5.** A functor  $F: \mathbf{D}^{op} \times \mathbf{D}' \rightarrow \mathbf{E}$ , contravariant in its first, covariant in its second variable, is called *locally continuous*, if for directed sets  $A \subseteq \text{Hom}(D_2, D_1)$  and  $A' \subseteq \text{Hom}(D'_1, D'_2)$  (where  $D_1, D_2$  are objects in  $\mathbf{D}$  and  $D'_1, D'_2$  are objects in  $\mathbf{D}'$ ) we have

$$F(\bigsqcup^\uparrow A, \bigsqcup^\uparrow A') = \bigsqcup_{f \in A, f' \in A'}^\uparrow F(f, f')$$

in  $\text{Hom}(F(D_1, D'_1), F(D_2, D'_2))$ .

**Proposition 5.2.6.** If  $F: \mathbf{D}^{op} \times \mathbf{D}' \rightarrow \mathbf{E}$  is a mixed variant, locally continuous functor, then it defines a continuous covariant functor  $\hat{F}$  from  $\mathbf{D}^e \times \mathbf{D}'^e$  to  $\mathbf{E}^e$  as follows:

$$\hat{F}(D, D') = F(D, D') \text{ for objects, and}$$

$$\hat{F}(e, e') = F(e^*, e') \text{ for embeddings.}$$

The upper adjoint to  $\hat{F}(e, e')$  is given by  $F(e, e'^*)$ .

**Proof.** Let  $(e, e^*)$  and  $(e', e'^*)$  be e-p-pairs in  $\mathbf{D}$  and  $\mathbf{D}'$ , respectively. We calculate  $F(e, e'^*) \circ \hat{F}(e, e') = F(e, e'^*) \circ F(e^*, e') = F(e^* \circ e, e'^* \circ e') = F(\text{id}, \text{id}) = \text{id}$  and  $\hat{F}(e, e') \circ F(e, e'^*) = F(e^*, e') \circ F(e, e'^*) = F(e \circ e^*, e' \circ e'^*) \subseteq F(\text{id}, \text{id}) = \text{id}$ , so  $\hat{F}$  maps indeed pairs of embeddings to embeddings.

For continuity, let  $\langle (D_n, (e_{mn})) \rangle$  and  $\langle (D'_n, (e'_{mn})) \rangle$  be expanding sequences in  $\mathbf{D}$  and  $\mathbf{D}'$  with colimits  $\langle D, (e_n) \rangle$  and  $\langle D', (e'_n) \rangle$ , respectively. By Lemma 3.3.8 this implies  $\bigsqcup_{n \in \mathbb{N}} e_n \circ e_n^* = \text{id}_D$  and  $\bigsqcup_{n \in \mathbb{N}} e'_n \circ e'^*_n = \text{id}_{D'}$ . By local continuity we have  $\bigsqcup_{n \in \mathbb{N}} \hat{F}(e_n, e'_n) \circ \hat{F}(e_n, e'_n)^* = \bigsqcup_{n \in \mathbb{N}} F(e_n^*, e'_n) \circ F(e_n, e'^*_n) = \bigsqcup_{n \in \mathbb{N}} F(e_n \circ e_n^*, e'_n \circ e'^*_n) = F(\bigsqcup_{n \in \mathbb{N}} e_n \circ e_n^*, \bigsqcup_{n \in \mathbb{N}} e'_n \circ e'^*_n) = F(\text{id}_D, \text{id}_{D'}) = \text{id}_{F(D, D')}$  and so  $\langle \hat{F}(D, D'), (\hat{F}(e_n, e'_n))_{n \in \mathbb{N}} \rangle$  is a colimit of  $\langle (\hat{F}(D_n, D'_n))_{n \in \mathbb{N}}, (\hat{F}(e_{mn}, e'_{mn}))_{n \in \mathbb{N}, m \in \mathbb{N}} \rangle$ . ■

While it may seem harmless to restrict a covariant functor to embeddings in order to solve a recursive domain equation, it is nevertheless not clear what the philosophical justification for this step is. For mixed variant functors this question becomes even more pressing since we explicitly *change* the functor. As already mentioned, a satisfactory answer has only recently been found [Freyd, 1991; Pitts, 1993a]. We present Peter Freyds solution in the next section.

Let us take stock of what we have achieved so far. Building blocks for recursive domain equations are the constructors of Section 3.2,  $\times, \oplus, \rightarrow$ , etc., each of which is readily seen to define a locally continuous functor. Translating them to embeddings via the preceding proposition, we get continuous functors of one or two variables. We further need the diagonal  $\Delta: \mathbf{D}^e \rightarrow \mathbf{D}^e \times \mathbf{D}^e$  to deal with multiple occurrences of  $X$  in the body of the equation. Then we note that colimits in a finite power of  $\mathbf{D}^e$  are calculated coordinatewise and hence the diagonal and the tupling of continuous functors are continuous. Finally, we include constant functors to allow for constants to occur in an equation. Two more operators will be added below: the bilimit in the next section and various powerdomain constructions in Chapter 6.

### 5.2.3 Parameterized equations

Suppose that we are given a locally continuous functor  $F$  in two variables. Given any domain  $D$  we can solve the equation  $X \cong F(D, X)$  using the techniques of the preceding sections. Remember that by default we mean the solution according to Lemma 5.2.2 based on  $e: \mathbb{I} \rightarrow F(D, \mathbb{I})$ , so there is no ambiguity. Also, we have given a concrete representation for bilimits in Theorem 3.3.7, so  $\text{FIX}(F(D, \cdot))$  is also well-defined in this respect. We want to show that it extends to a functor.

Notation is a bit of a problem. Let  $F: \mathbf{D}_{\perp!} \times \mathbf{E}_{\perp!} \rightarrow \mathbf{E}_{\perp!}$  be a functor in two variables. We set  $F_D$  for the functor on  $\mathbf{E}_{\perp!}$  which maps  $E$  to  $F(D, E)$  for objects and  $g: E \xrightarrow{\perp!} E'$  to  $F(\text{id}_D, g)$  for morphisms. Similarly for  $F_{D'}$ . The embeddings into the canonical fixpoint of  $F_D$ , resp.  $F_{D'}$ , we denote by  $e_0, e_1, \dots$  and  $e'_0, e'_1, \dots$ , and we use  $e$  and  $e'$  for the unique strict function from  $\mathbb{I}$  into  $D$  and  $D'$ , respectively.

**Proposition 5.2.7.** *Let  $F: \mathbf{D}_{\perp!} \times \mathbf{E}_{\perp!} \rightarrow \mathbf{E}_{\perp!}$  be a locally continuous functor. Then the following defines a locally continuous functor from  $\mathbf{D}_{\perp!}$  to  $\mathbf{E}_{\perp!}$ :*

$$\begin{aligned} \text{On objects} &: D \mapsto \text{FIX}(F_D), \\ \text{on morphisms} &: (f: D \rightarrow D') \mapsto \bigsqcup_{n \in \mathbb{N}} \uparrow e'_n \circ f_n \circ e_n^* \end{aligned}$$

where the sequence  $(f_n)_{n \in \mathbb{N}}$  is defined recursively by  $f_0 = \text{id}_{\mathbb{I}}$ ,  $f_{n+1} = F(f, f_n)$ .

**Proof.** Let  $D$  and  $D'$  be objects of  $\mathbf{D}_{\perp!}$  and let  $f: D \xrightarrow{\perp!} D'$  be a strict function. The solution to  $X \cong F(D, X)$  is given by the bilimit

$$\begin{array}{ccccccc} & & \text{FIX}(F_D) & & & & \\ & \nearrow e_0 & \uparrow e_1 & \nwarrow e_2 & \dots & & \\ \mathbb{I} & \xrightarrow{e} & F_D(\mathbb{I}) & \xrightarrow{F_D(e)} & F_D^2(\mathbb{I}) & \longrightarrow & \dots \end{array}$$

and similarly for  $D'$ . Corresponding objects of the two expanding sequences are connected by  $f_n: F_D^n(\mathbb{I}) \xrightarrow{\perp!} F_{D'}^n(\mathbb{I})$ . They commute with the embeddings of the expanding sequences: For  $n = 0$  we have  $F_{D'}^0(e') \circ f_0 = e' \circ \text{id}_{\mathbb{I}} = e' = f_1 \circ e = f_1 \circ F_D^0(e)$  because there is only one strict map from  $\mathbb{I}$  to  $F^1(D')$ . Higher indices follow by induction:

$$\begin{aligned} F_{D'}^{n+1}(e') \circ f_{n+1} &= F(\text{id}_{D'}, F_{D'}^n(e')) \circ F(f, f_n) \\ &= F(f, F_{D'}^n(e') \circ f_n) \\ &= F(f, f_{n+1} \circ F_D^n(e)) \\ &= F(f, f_{n+1}) \circ F(\text{id}_D, F_D^n(e)) \\ &= f_{n+2} \circ F_D^{n+1}(e). \end{aligned}$$

So we have a second cocone over the sequence defining  $\text{FIX}(F_D)$  and using the fact that colimits in  $\mathbf{E}_{\perp!}^e$  are also colimits in  $\mathbf{E}_{\perp!}$ , we get a (unique) mediating morphism from  $\text{FIX}(F_D)$  to  $\text{FIX}(F_{D'})$ . By Theorem 3.3.7 it has the postulated representation.

Functoriality comes free from the uniqueness of mediating morphisms. It remains to check local continuity. So let  $A$  be a directed set of maps from  $D$  to  $D'$ . We easily get  $(\bigsqcup^1 A)_n = \bigsqcup^1_{f \in A} f_n$  by induction and the local continuity of  $F$ . The supremum can be brought to the very front by the continuity of composition and general associativity. ■

Note that this proof works just as well for mixed variant functors. As an application, suppose we are given a system of simultaneous equations

$$\begin{array}{ccc} X_1 & \cong & F_1(X_1, \dots, X_n) \\ & \vdots & \\ X_n & \cong & F_n(X_1, \dots, X_n). \end{array}$$

We can solve these one after the other, viewing  $X_2, \dots, X_n$  as parameters for the first equation, substituting the result for  $X_1$  in the second equation and so on. It is more direct to pass from  $\mathbf{D}$  to  $\mathbf{D}^n$ , for which Theorem 3.3.7 and the results of this chapter remain true, and then solve these equations simultaneously with the tupling of the  $F_i$ . The fact that these two methods yield isomorphic results is known as *Bekič's rule* [Bekič, 1969].

## 5.3 Canonicity

We have seen in the first section of this chapter that recursive domain equations arise in various contexts. Having demonstrated a technique for solving them, we must now check whether the solutions match the particular requirements of these applications.

### 5.3.1 Invariance and minimality

Let us begin with a technique of internalizing the expanding sequence  $\mathbb{I} \rightarrow \mathbb{F}(\mathbb{I}) \rightarrow \mathbb{F}(\mathbb{F}(\mathbb{I})) \rightarrow \dots$  into the canonical solution. This will allow us to do proofs about  $\text{FIX}(F)$  without (explicit) use of the defining expanding sequence.

**Lemma 5.3.1.** *Let  $F$  be a locally continuous functor on a category of domains  $\mathbf{D}$  and let  $i: F(A) \rightarrow A$  be an isomorphism. Then there exists a least homomorphism  $h_{C,A}$  from  $A$  to every other  $F$ -algebra  $f: F(C) \rightarrow C$ . It equals the least fixpoint of the functional  $\phi_{C,A}$  on  $[A \rightarrow C]$  which is defined by*

$$\phi_{C,A}(g) = f \circ F(g) \circ i^{-1}.$$

*Least homomorphisms compose: If  $j: F(B) \rightarrow B$  is also an isomorphism, then  $h_{C,A} = h_{C,B} \circ h_{B,A}$ .*

**Proof.** The functional  $\phi = \phi_{C,A}$  is clearly continuous because  $F$  is locally continuous and composition is a continuous operation. Since we have globally assumed least elements, the function space  $[A \rightarrow C]$  contains  $c_\perp$  as a



least element. So the least fixpoint  $h_{C,A}$  of  $\phi_{C,A}$  calculated as the supremum of the chain  $c_\perp \sqsubseteq \phi(c_\perp) \sqsubseteq \dots$  exists. We show by induction that it is below every homomorphism  $h$ . For  $c_\perp$  this is obvious. For the induction step assume  $g \sqsubseteq h$ . We calculate:  $\phi(g) = f \circ F(g) \circ i^{-1} \sqsubseteq f \circ F(h) \circ i^{-1} = h$ . It follows that  $\text{fix}(\phi) = h_{C,A} \sqsubseteq h$  holds. On the other hand, every fixpoint of  $\phi$  is a homomorphism:  $h \circ i = \phi(h) \circ i = f \circ F(h) \circ i^{-1} \circ i = f \circ F(h)$ .

The claim about composition of least homomorphisms can also be shown by induction. But it is somewhat more elegant to use the invariance of least fixpoints, Lemma 2.1.21. Consider the diagram

$$\begin{array}{ccc} [B \longrightarrow C] & \xrightarrow{H} & [A \longrightarrow C] \\ \downarrow \phi_{C,B} & & \downarrow \phi_{C,A} \\ [B \longrightarrow C] & \xrightarrow{H} & [A \longrightarrow C] \end{array}$$

where  $H$  is the strict operation which assigns  $g \circ h_{B,A}$  to  $g \in [B \longrightarrow C]$ . The diagram commutes, because  $H \circ \phi_{C,B}(g) = f \circ F(g) \circ j^{-1} \circ h_{B,A} = f \circ F(g \circ h_{B,A}) \circ i^{-1}$  (because  $h_{B,A}$  is an homomorphism)  $= \phi_{C,A}(H(g))$ . Lemma 2.1.21 then gives us the desired equality:  $h_{C,A} = \text{fix}(\phi_{C,A}) = H(\text{fix}(\phi_{C,B})) = \text{fix}(\phi_{C,B}) \circ h_{B,A} = h_{C,B} \circ h_{B,A}$ . ■

Specializing the second algebra in this lemma to be  $i: F(A) \rightarrow A$  itself, we deduce that on every fixpoint of a locally continuous functor there exists a least endomorphism  $h_{A,A}$ . Since the identity is always an endomorphism, the least endomorphism must be below the identity and idempotent, i.e. a kernel operator and in particular strict. This we will use frequently below.

**Theorem 5.3.2 (Invariance, Part 1).** *Let  $F$  be a locally continuous functor on a category of domains  $\mathbf{D}$  and let  $i: F(A) \rightarrow A$  be an isomorphism. Then the following are equivalent:*

1.  $A$  is isomorphic to the canonical fixpoint  $\text{FIX}(F)$ ;
2.  $\text{id}_A$  is the least endomorphism of  $A$ ;
3.  $\text{id}_A = \text{fix}(\phi_{A,A})$  where  $\phi_{A,A}: [A \longrightarrow A] \rightarrow [A \longrightarrow A]$  is defined by  $\phi_{A,A}(g) = i \circ F(g) \circ i^{-1}$ ;
4.  $\text{id}_A$  is the only strict endomorphism of  $A$ .

**Proof.** (1  $\implies$  2) The least endomorphism on  $D = \text{FIX}(F)$  is calculated as the least fixpoint of  $\phi_{D,D}: g \mapsto \text{fold} \circ F(g) \circ \text{unfold}$ . With the usual notation for the embeddings of  $F^n(\mathbb{I})$  into  $D$  we get (by induction):  $c_\perp = e_0 \circ e_0^*$  and  $\phi^n(c_\perp) = \phi(\phi^{n-1}(c_\perp)) = \phi(e_{n-1} \circ e_{n-1}^*) = \text{fold} \circ F(e_{n-1}) \circ F(e_{n-1}^*) \circ \text{unfold} = e_n \circ e_n^*$ , where the last equality follows because  $\text{fold}$  and  $\text{unfold}$  are mediating morphisms. Lemma 3.3.8 entails that the supremum of the  $\phi^n(c_\perp)$  is the identity.

The equivalence of (2) and (3) is a reformulation of Lemma 5.3.1.

(3  $\implies$  4) Suppose  $h: A \xrightarrow{\perp\perp} A$  defines an endomorphism of the algebra  $i: F(A) \rightarrow A$ . We apply the invariance property of least fixpoints, Lemma 2.1.21, to the diagram (where  $\phi$  now stands for  $\phi_{A,A}$ )

$$\begin{array}{ccc} [A \longrightarrow A] & \xrightarrow{H} & [A \longrightarrow A] \\ \downarrow \phi & & \downarrow \phi \\ [A \longrightarrow A] & \xrightarrow{H} & [A \longrightarrow A] \end{array}$$

where  $H$  maps  $g \in [A \longrightarrow A]$  to  $h \circ g$ . This is a strict operation because  $h$  is assumed to be strict. The diagram commutes:  $H \circ \phi(g) = H(i \circ F(g) \circ i^{-1}) = h \circ i \circ F(g) \circ i^{-1} = i \circ F(h) \circ F(g) \circ i^{-1} = \phi(H(g))$ . By Lemma 2.1.21 we have  $\text{id}_A = \text{fix}(\phi) = H(\text{fix}(\phi)) = h \circ \text{id}_A = h$ .

(4  $\implies$  1) By the preceding lemma we have homomorphisms between  $A$  and  $\text{FIX}(F)$ . They compose to the least endomorphisms on  $A$ , resp.  $\text{FIX}(F)$ , which we know to be strict. But then they must be equal to the identity as we have just shown for  $\text{FIX}(F)$  and assumed for  $A$ . ■

If, in the last third of this proof, we do not assume that  $\text{id}_A$  is the only strict endomorphism on  $A$ , then we still get an embedding–projection pair between  $\text{FIX}(F)$  and  $A$ . Thus we have:

**Theorem 5.3.3.** (Minimality, Part 1) *The canonical fixpoint of a locally continuous functor is a sub-domain of every other fixpoint.*

So we have shown that the canonical solution is the *least* fixpoint in a relevant sense. This is clearly a good canonicity result with respect to the first class of examples. For pedagogical reasons we have restricted attention to the covariant case first, but, as we will see in Section 5.3.3, this characterization is also true for functors of mixed variance.

### 5.3.2 Initiality and finality

By a little refinement of the proofs of the preceding subsection we get the desired result that the canonical fixpoint together with fold is an initial  $F$ -algebra. One of the adjustments is that we have to pass completely to strict functions, because Lemma 5.3.1 does not guarantee the existence of strict homomorphisms, and only of these can we prove unicity.

**Theorem 5.3.4 (Initiality).** *Let  $F: \mathbf{D}_{\perp\perp} \rightarrow \mathbf{D}_{\perp\perp}$  be a locally continuous functor on a category of domains with strict functions. Then  $\text{fold}: F(D) \rightarrow D$  is an initial  $F$ -algebra where  $D$  is the canonical solution to  $X \cong F(X)$ .*

**Proof.** Let  $f: F(A) \xrightarrow{\perp\perp} A$  be a strict  $F$ -algebra. The homomorphism  $h: D \rightarrow A$  we get from Lemma 5.3.1 is strict as we see by inspecting its def-

inition. That there are no others is shown as in the proof of Theorem 5.3.2 ( $2 \implies 3$ ). The relevant diagram for the application of Lemma 2.1.21 is now:

$$\begin{array}{ccc}
 [D \rightarrow D] & \xrightarrow{H} & [D \rightarrow A] \\
 \phi_{D,D} \downarrow & & \downarrow \phi_{A,D} \\
 [D \rightarrow D] & \xrightarrow{H} & [D \rightarrow A].
 \end{array}$$

■

By dualizing Lemma 5.3.1 and the proof of Theorem 5.3.2, ( $2 \implies 3$ ), we get the final co-algebra theorem. It is slightly stronger than initiality since it holds for all co-algebras, not only the strict ones.

**Theorem 5.3.5 (Finality).** *Let  $F: \mathbf{D} \rightarrow \mathbf{D}$  be a locally continuous functor with canonical fixpoint  $D = \text{FIX}(F)$ . Then  $\text{unfold}: D \rightarrow F(D)$  is a final co-algebra.*

### 5.3.3 Mixed variance

Let us now tackle the case that we are given an equation in which the variable  $X$  occurs both positively and negatively in the body, as in our first example  $X \cong [X \rightarrow X]$ . We assume that by separating the negative occurrences from the positive ones, we have a functor in two variables, contravariant in the first and covariant in the second. As the reader will remember, solving such an equation required the somewhat magical passage to adjoints in the first coordinate. We will now see how far we can extend the results from the previous two subsections to this case. Note that for a mixed variant functor the concept of  $F$ -algebra or co-algebra is no longer meaningful, as there are no homomorphisms. The idea is to pass to pairs of mappings. Lemma 5.3.1 is replaced by

**Lemma 5.3.6.** *Let  $F: \mathbf{D}^{op} \times \mathbf{D} \rightarrow \mathbf{D}$  be a mixed variant, locally continuous functor and let  $i: F(A, A) \rightarrow A$  and  $j: F(B, B) \rightarrow B$  be isomorphisms. Then there exists a least pair of functions  $h: A \rightarrow B$  and  $k: B \rightarrow A$  such that*

$$\begin{array}{ccccc}
 F(A, A) & \xrightarrow{F(k, h)} & F(B, B) & & F(B, B) & \xrightarrow{F(h, k)} & F(A, A) \\
 i \downarrow & & \downarrow j & \text{and} & j \downarrow & & \downarrow i \\
 A & \xrightarrow{h} & B & & B & \xrightarrow{k} & A
 \end{array}$$

commute.

*The composition of two such least pairs gives another one.*

**Proof.** Define a Scott-continuous function  $\phi$  on  $[A \rightarrow B] \times [B \rightarrow A]$  by  $\phi(f, g) = (j \circ F(g, f) \circ i^{-1}, i \circ F(f, g) \circ j^{-1})$  and let  $(h, k)$  be its least fixpoint. Commutativity of the two diagrams is shown as in the proof of Lemma 5.3.1. ■

By equating  $A$  and  $B$  in this lemma, we get a least endofunction  $h$  which satisfies  $h \circ f = f \circ F(h, h)$ . Again, it must be below the identity. Let us call such endofunctions *mixed endomorphisms*.

**Theorem 5.3.7 (Invariance, Part 2).** *Let  $F: \mathbf{D}^{op} \times \mathbf{D} \rightarrow \mathbf{D}$  be a mixed variant and locally continuous functor and let  $i: F(A, A) \rightarrow A$  be an isomorphism. Then the following are equivalent:*

1.  $A$  is isomorphic to the canonical fixpoint  $\text{FIX}(F)$ ;
2.  $\text{id}_A$  is the least mixed endomorphism of  $A$ ;
3.  $\text{id}_A = \text{fix}(\phi_{A,A})$  where  $\phi_{A,A}: [A \rightarrow A] \rightarrow [A \rightarrow A]$  is defined by  $\phi_{A,A}(g) = i \circ F(g, g) \circ i^{-1}$ ;
4.  $\text{id}_A$  is the only strict mixed endomorphism of  $A$ .

**Proof.** The proof is of course similar to that of Theorem 5.3.2, but let us spell out the parts where mixed variance shows up. Recall from Section 5.2.2 how the expanding sequence defining  $D = \text{FIX}(F)$  looks like:  $\mathbb{I} \xrightarrow{e} \mathbb{F}(\mathbb{I}, \mathbb{I}) \xrightarrow{F(e^*, e)} \mathbb{F}(\mathbb{F}(\mathbb{I}, \mathbb{I}), \mathbb{F}(\mathbb{I}, \mathbb{I})) \longrightarrow \dots$ . If  $e_0, e_1, \dots$  are the colimiting maps into  $D$ , then  $F(e_0^*, e_0), F(e_1^*, e_1), \dots$  form the cocone into  $F(D, D)$ , which, by local continuity, is also colimiting. The equations from Lemma 5.2.2 read:  $F(e_n^*, e_n) = \text{unfold} \circ e_{n+1}$  and  $F(e_n^*, e_n)^* = F(e_n, e_n^*) = e_{n+1}^* \circ \text{fold}$ . We show that the  $n$ th approximation to the least mixed endomorphism equals  $e_n \circ e_n^*$ . For  $n = 0$  we get  $c_\perp = e_0 \circ e_0^*$ , and for the induction step:

$$\begin{aligned}
 \phi^{n+1}(c_\perp) &= \phi(\phi^n(c_\perp)) \\
 &= \phi(e_n \circ e_n^*) \\
 &= \text{fold} \circ F(e_n \circ e_n^*, e_n \circ e_n^*) \circ \text{unfold} \\
 &= \text{fold} \circ F(e_n^*, e_n) \circ F(e_n, e_n^*) \circ \text{unfold} \\
 &= e_{n+1} \circ e_{n+1}^*.
 \end{aligned}$$

(Note how contravariance in the first argument of  $F$  shuffles  $e_n$  and  $e_n^*$  in just the right way.)

(3  $\implies$  4) The diagram to which Lemma 2.1.21 is applied is as before, but  $H: [A \rightarrow A] \rightarrow [A \rightarrow A]$  now maps  $g: A \rightarrow A$  to  $h \circ g \circ h$ .

The rest can safely be left to the reader. ■

**Theorem 5.3.8 (Minimality, Part 2).** *The canonical fixpoint of a mixed variant and locally continuous functor is a sub-domain of every other fixpoint.*

Now that we have some experience with mixed variance, it is pretty clear how to deal with initiality and finality. The trick is to pass once more to pairs of (strict) functions.

**Theorem 5.3.9 (Free mixed variant algebra).** *Let  $F: \mathbf{D}_{\perp!}^{op} \times \mathbf{D}_{\perp!} \rightarrow \mathbf{D}_{\perp!}$  be a mixed variant, locally continuous functor and let  $D$  be the canonical solution to  $X \cong F(X, X)$ . Then for every pair of strict continuous functions  $f: A \xrightarrow{\perp!} F(B, A)$  and  $g: F(A, B) \xrightarrow{\perp!} B$  there are unique strict functions  $h: A \xrightarrow{\perp!} D$  and  $k: D \xrightarrow{\perp!} B$  such that*

$$\begin{array}{ccccc}
 F(B, A) & \xrightarrow{F(k, h)} & F(D, D) & \xrightarrow{F(h, k)} & F(A, B) \\
 \uparrow f & & \uparrow \text{unfold and} & & \downarrow g \\
 A & \xrightarrow{h} & D & \xrightarrow{k} & B \\
 & & \downarrow \text{fold} & & \\
 & & D & & 
 \end{array}$$

commute.

We should mention that the passage from covariant to mixed-variant functors, which we have carried out here concretely, can be done on an abstract, categorical level as was demonstrated by Peter Freyd in [Freyd, 1991]. The feature of domain theory which Freyd uses as his sole axiom is the existence and coincidence of initial algebras and final co-algebras for ‘all’ endofunctors (‘all’ to be interpreted in some suitable enriched sense, in our case as ‘all locally continuous endofunctors’). Freyds results are the most striking contribution to date towards axiomatic domain theory, for which see Section 8.4 below.

## 5.4 Analysis of solutions

We have worked hard in the last section in order to show that our domain-theoretic solutions are canonical in various respects. Besides this being reassuring, the advantage of canonical solutions is that we can establish proof rules for showing properties of them. This is the topic of this section.

### 5.4.1 Structural induction on terms

This technique is in analogy with universal algebra. While one has no control over arbitrary algebras of a certain signature, we feel quite comfortable with the initial or term algebra. There, every element is described by a term and no identifications are made. The first property carries over to our setting quite easily. For each of the finitary constructions of Section 3.2,



we have introduced a notation for the basis elements of the constructed domain, to wit, tuples  $\langle d, e \rangle$ , variants  $(d : i)$ , one-element constant  $\perp \in \mathbb{I}$ , and step-functions  $(d \searrow e)$ . Since our canonical solutions are built as bilimits, starting from  $\mathbb{I}$ , and since every basis element of a bilimit shows up at a finite iteration, Theorem 3.3.11, these can be denoted by finite expressions. The proof can then be based on structural induction on the length of these terms.

Unicity, however, is hard to achieve and this is the fault of the function space. One has to define normal forms and prove conversion rules. A treatment along these lines, based on [Abramsky, 1991b], is given in Chapter 7.3.

### 5.4.2 Admissible relations

This is a more domain-theoretic formulation of structural induction, based on certain relations. The subject has recently been expanded and re-organized in an elegant way by Andrew Pitts [1992; 1993a]. We follow his treatment closely but do not seek the same generality. We start with admissible relations, which we have met shortly in Chapter 2 already.

**Definition 5.4.1.** A relation  $R \subseteq D^n$  on a pointed domain  $D$  is called *admissible* if it contains the constantly bottom tuple and if it is closed under suprema of  $\omega$ -chains. We write  $\mathcal{R}^n(D)$  for the set of all admissible  $n$ -ary relations on  $D$ , ordered by inclusion. Unary relations of this kind are also called *admissible predicates*.

This is tailored to applications of Theorem 2.1.19, whence we preferred the slightly more inclusive concept of  $\omega$ -chain over directed sets. If we are given a strict continuous function  $f: D \xrightarrow{\perp!} E$ , then we can apply it to relations pointwise in the usual way:

$$f^{rel}(R) = \{ \langle f(x_1), \dots, f(x_n) \rangle \mid \langle x_1, \dots, x_n \rangle \in R \}.$$

**Proposition 5.4.2.** For dcpos  $D$  and  $E$  and admissible  $n$ -ary relations  $R$  on  $D$  and  $S$  on  $E$  the set  $\{f \mid f^{rel}(R) \subseteq S\}$  is an admissible predicate on  $[D \xrightarrow{\perp!} E]$ .

We also need to say how admissible relations may be transformed by our locally continuous functors. This is a matter of definition because there are several—and equally useful—possibilities.

**Definition 5.4.3.** Let  $F: \mathbf{D}_{\perp!}^{op} \times \mathbf{D}_{\perp!} \rightarrow \mathbf{D}_{\perp!}$  be a mixed variant and locally continuous functor on a category of domains and strict functions. An *admissible action* on ( $n$ -ary) relations for  $F$  is given by a function  $F^{rel}$  which assigns to each pair  $\langle D, E \rangle$  a map  $F_{(D,E)}^{rel}$  from  $\mathcal{R}(D) \times \mathcal{R}(E)$  to  $\mathcal{R}(F(D, E))$ . These maps have to be compatible with strict morphisms in  $\mathbf{D}_{\perp!}$  as follows: If  $f: D_2 \xrightarrow{\perp!} D_1$  and  $g: E_1 \xrightarrow{\perp!} E_2$  and if  $R_1 \in \mathcal{R}(D_1)$

etc., such that  $f^{rel}(R_2) \subseteq R_1$  and  $g^{rel}(S_1) \subseteq S_2$ , then

$$F(f, g)^{rel}(F_{\langle D_1, E_1 \rangle}^{rel}(R_1, S_1)) \subseteq F_{\langle D_2, E_2 \rangle}^{rel}(R_2, S_2).$$

(Admittedly, this is a bit heavy in terms of notation. But in our concrete examples it is simply not the case that the behaviour of  $F_{\langle D, E \rangle}^{rel}$  on  $R$  and  $S$  is the same as—or in a simple way related to—the result of applying the functor to  $R$  and  $S$  viewed as dcpos.)

Specializing  $f$  and  $g$  to identity mappings in this definition, we obtain

**Proposition 5.4.4.** *The maps  $F_{\langle D, E \rangle}^{rel}$  are antitone in the first and monotone in the second variable.*

**Theorem 5.4.5.** *Let  $\mathbf{D}_{\perp!}$  be a category of domains and let  $F$  be a mixed variant and locally continuous functor from  $\mathbf{D}_{\perp!}^{op} \times \mathbf{D}_{\perp!}$  to  $\mathbf{D}_{\perp!}$  together with an admissible action on relations. Abbreviate  $\text{FIX}(F)$  by  $D$ . Given two admissible relations  $R, S \in \mathcal{R}^n(D)$  such that*

$$\text{unfold}^{rel}(R) \subseteq F^{rel}(S, R) \quad \text{and} \quad \text{fold}^{rel}(F^{rel}(R, S)) \subseteq S$$

*then  $R \subseteq S$  holds.*

**Proof.** We know from the invariance theorem that the identity on  $D$  is the least fixpoint of  $\phi$ , where  $\phi(g) = \text{fold} \circ F(g, g) \circ \text{unfold}$ . Let  $P = \{f \in [D \xrightarrow{\perp!} D] \mid f^{rel}(R) \subseteq S\}$ , which we know is an admissible predicate. We want that the identity on  $D$  belongs to  $P$  and for this it suffices to show that  $\phi$  maps  $P$  into itself. So suppose  $g \in P$ :

$$\begin{aligned} \phi(g)^{rel}(R) &= \text{fold}^{rel} \circ F(g, g)^{rel} \circ \text{unfold}^{rel}(R) && \text{by definition} \\ &\subseteq \text{fold}^{rel} \circ F(g, g)^{rel}(F^{rel}(S, R)) && \text{by assumption} \\ &\subseteq \text{fold}^{rel}(F^{rel}(R, S)) && \text{because } g \in P \\ &\subseteq S && \text{by assumption} \end{aligned}$$

Indeed,  $\phi(g)$  belongs again to  $P$ . ■

In order to understand the power of this theorem, we will study two particular actions in the next subsections. They, too, are taken from [Pitts, 1993a].

### 5.4.3 Induction with admissible relations

**Definition 5.4.6.** Let  $F$  be a mixed variant functor as before. We call an admissible action on ( $n$ -ary) relations *logical* if, for all objects  $D$  and  $E$  and  $R \in \mathcal{R}^n(D)$ , we have  $F_{\langle D, E \rangle}^{rel}(R, E^n) = F(D, E)^n$ .

Specializing  $R$  to be the whole  $D$  in Theorem 5.4.5 and removing the assumption  $\text{unfold}^{rel}(R) \subseteq F^{rel}(S, R)$ , which for this choice of  $R$  is always satisfied for a logical action, we obtain

**Theorem 5.4.7 (Induction).** *Let  $\mathbf{D}_{\perp!}$  be a category of domains and let  $F: \mathbf{D}_{\perp!}^{op} \times \mathbf{D}_{\perp!} \rightarrow \mathbf{D}_{\perp!}$  be a mixed variant and locally continuous functor together with a logical action on admissible predicates. Let  $D$  be the canonical fixpoint of  $F$ . If  $S \in \mathcal{R}^1(D)$  is an admissible predicate, for which  $x \in F^{rel}(D, S)$  implies  $\text{fold}(x) \in S$ , then  $S$  must be equal to  $D$ .*

The reader should take the time to recognize in this the principle of structural induction on term algebras.

We exhibit a particular logical action on admissible predicates for functors which are built from the constructors of Section 3.2. If  $R, S$  are admissible predicates on the pointed domains  $D$  and  $E$ , then we set

$$\begin{aligned} R_{\perp} &= \text{up}(R) \cup \{\perp\} \subseteq D, \\ R \times S &= \{\langle x, y \rangle \in D \times E \mid x \in R, y \in S\}, \\ [R \longrightarrow S] &= \{f \in [D \longrightarrow E] \mid f(R) \subseteq S\}, \\ R \oplus S &= \text{inl}(R) \cup \text{inr}(S) \subseteq D \oplus E, \end{aligned}$$

and analogously for  $\otimes$  and  $[\cdot \xrightarrow{\perp!} \cdot]$ . (This is not quite in accordance with our notational convention. For example, the correct expression for  $[R \longrightarrow S]$  is  $[\cdot \longrightarrow \cdot]_{\langle D, E \rangle}^{rel}(R, S)$ .) The definition of the action for the function space operator should make it clear why we chose the adjective ‘logical’ for it.

We get more complicated functors by composing the basic constructors. The actions also compose in a straightforward way: If  $F$ ,  $G_1$ , and  $G_2$  are mixed variant functors on a category of domains, then we can define a mixed variant composition  $H = F \circ \langle G_1, G_2 \rangle$  by setting  $H(X, Y) = F(G_1(Y, X), G_2(X, Y))$  for objects and similarly for morphisms. Given admissible actions for each of  $F$ ,  $G_1$ , and  $G_2$ , we can define an action for  $H$  by setting  $H^{rel}(R, S) = F^{rel}(G_1^{rel}(S, R), G_2^{rel}(R, S))$ . It is an easy exercise to show that this action is logical if all its constituents are.

#### 5.4.4 Co-induction with admissible relations

In this subsection we work with another canonical relation on domains, namely the order relation. We again require that it is dominant if put in the covariant position.

**Definition 5.4.8.** Let  $F$  be a mixed variant functor. We call an admissible action on binary relations *extensional*, if for all objects  $D$  and  $E$  and  $R \in \mathcal{R}^n(D)$  we have  $F_{(D, E)}^{rel}(R, \sqsubseteq_E) = \sqsubseteq_{F(D, E)}$ .

**Theorem 5.4.9.** (Co-induction) *Let  $\mathbf{D}_{\perp!}$  be a category of domains and let  $F: \mathbf{D}_{\perp!}^{op} \times \mathbf{D}_{\perp!} \rightarrow \mathbf{D}_{\perp!}$  be a mixed variant and locally continuous functor together with an extensional action on binary relations. Let  $D$  be the canonical fixpoint of  $F$ . If  $R \in \mathcal{R}^1(D)$  is an admissible relation such that for all  $\langle x, y \rangle \in R$  we have  $\langle \text{unfold}(x), \text{unfold}(y) \rangle \in F^{rel}(\sqsubseteq_D, R)$ , then  $R$  is*

contained in  $\sqsubseteq_D$ .

If we call an admissible binary relation  $R$  on  $D$  a *simulation* if it satisfies the hypothesis of this theorem, then we can formulate quite concisely:

**Corollary 5.4.10.** *Two elements of the canonical fixpoint of a mixed variant and locally continuous functor are in the order relation if and only if they are related by a simulation.*

We still have to show that extensional actions exist. We proceed as in the last subsection and first give extensional actions for the primitive constructors and then rely on the fact that these compose. So let  $R, S$  be admissible binary relations on  $D$ , resp.  $E$ . We set:

$$\begin{aligned}
 R_{\perp} &= \{\langle x, y \rangle \in D^2 \mid x = \perp \text{ or } \langle x, y \rangle \in R\} \\
 R \times S &= \{\langle \langle x, y \rangle, \langle x', y' \rangle \rangle \in (D \times E)^2 \mid \\
 &\quad \langle x, x' \rangle \in R \text{ and } \langle y, y' \rangle \in S\} \\
 [R \longrightarrow S] &= \{\langle f, g \rangle \in [D \longrightarrow E]^2 \mid \forall x \in D. \langle f(x), g(x) \rangle \in S\} \\
 R \oplus S &= \{\langle x, y \rangle \in (D \oplus E)^2 \mid x = \perp \text{ or } \\
 &\quad (x = \text{inl}(x'), y = \text{inl}(y') \text{ and } \langle x', y' \rangle \in R) \text{ or } \\
 &\quad (x = \text{inr}(x'), y = \text{inr}(y') \text{ and } \langle x', y' \rangle \in S)\}
 \end{aligned}$$

and similarly for  $\otimes$  and  $[\cdot \xrightarrow{\perp} \cdot]$ . We call this family of actions ‘extensional’ because the definition in the case of the function space is the same as for the extensional order on functions.

#### Exercises 5.4.11.

1. Find recursive domain equations which characterize the three versions of the natural numbers from Figure 2.
2. [Erné, 1985] Find an example which demonstrates that the ideal completion functor is not locally continuous. Characterize the solutions to  $X \cong \text{Idl}(X, \sqsubseteq)$ .
3. [Davies *et al.*, 1971] Prove that only the one-point poset satisfies  $P \cong [P \xrightarrow{m} P]$ .
4. Verify Bekiřs rule in the dcpo case. That is, let  $D, E$  be pointed dcpos and let  $f: D \times E \rightarrow D$  and  $g: D \times E \rightarrow E$  be continuous functions. We can solve the equations

$$x = f(x, y) \quad y = g(x, y)$$

directly by taking the simultaneous fixpoint  $(a, b) = \text{fix}(\langle f, g \rangle)$ . Or we can solve for one variable at a time by defining

$$h(y) = \text{fix}(\lambda x. f(x, y)) \quad k(y) = g(h(y), y)$$

and setting

$$d = \text{fix}(k) \quad c = h(d) .$$

Verify that  $(a, b) = (c, d)$  holds by using fixpoint induction.

5. Find an example which shows that Theorem 5.3.4 may fail for non-strict algebras.
6. Why does Theorem 5.3.5 hold for arbitrary (non-strict) co-algebras?
7. What are initial algebra and final co-algebra for the functor  $X \mapsto \mathbb{I} \dot{\cup} \mathbb{X}$  on the category of sets? Show that they are not isomorphic as algebras.
8. (G. Plotkin) Let  $F$  be the functor which maps  $X$  to  $[X \longrightarrow X]_{\perp}$  and let  $D$  be its canonical fixpoint. This gives rise to a model of the (lazy) lambda calculus (see [Barendregt, 1984; Abramsky, 1990c; Abramsky and Ong, 1993]). Prove that the denotation of the  $Y$  combinator in this model is the least fixpoint function  $\text{fix}$ . Proceed as follows:
  - (a) Define a multiplication on  $D$  by  $x \cdot y = \text{unfold}(x)(y)$ .
  - (b) The interpretation  $y_f$  of  $Yf$  is  $\omega_f \cdot \omega_f$  where  $\omega_f = \text{fold}(x \mapsto f(x \cdot x))$ . Check that this is a fixpoint of  $f$ . It follows that  $\text{fix}(f) \sqsubseteq y_f$  holds.
  - (c) Define a subset  $E$  of  $[D \longrightarrow D]_{\perp}$  by

$$E = \{e \mid e \sqsubseteq \text{id}_D \text{ and } e(\omega_f) \cdot \omega_f \sqsubseteq \text{fix}(f)\} .$$

- (d) Use Theorem 5.3.7 to show that  $\text{id}_D \in E$ . Then  $y_f \sqsubseteq \text{fix}(f)$  is also valid.
9. Given an action on relations for a functor in four variables, contravariant in the first two, covariant in the last two, define an action for the functor  $(D, E) \mapsto \text{FIX}(F(D, \cdot, E, \cdot))$ . Prove that the resulting action is logical (extensional) if the original action was logical (extensional).

## 6 Equational theories

In the last chapter we saw how we can build initial algebras over domains. It is a natural question to ask whether we can also accommodate equations, i.e. construct free algebras with respect to equational theories. In universal algebra this is done by factoring the initial or term algebra with respect to the congruence generated by the defining equations, and we will see that we can proceed in a similar fashion for domains. Bases will play a prominent role in this approach.

The technique of the previous chapter, namely, to generate the desired algebra in an iterative process, is no longer applicable. A formal proof for this statement may be found in [Adámek and Trnková, 1989], Section III.3, but the result is quite intuitive: Recall that an  $F$ -algebra  $\alpha: F(A) \rightarrow A$  encodes the algebraic structure on  $A$  by giving information about the basic



operations on  $A$ , where  $F(A)$  is the sum of the input domains for each basic operation. Call an equation *flat* if each of the equated terms contains precisely one operation symbol. For example, commutativity of a binary operation is expressed by a flat equation while associativity is not. Flat equations can be incorporated into the concept of  $F$ -algebras by including the input, on which the two operations agree, only once in  $F(A)$ . For non-flat equations such a trick is not available. What we need instead of just the basic operations is a description of all term operations over  $A$ . In this case,  $F(A)$  will have to be the free algebra over  $A$ , the object we wanted to construct!

Thus  $F$ -algebras are not the appropriate categorical concept to model equational theories. The correct formalization, rather, is that of monads and Eilenberg–Moore algebras.

We will show the existence of free algebras for dcpos and continuous domains in the first section of this chapter. For the former, we use the adjoint functor theorem (see [Poigné, 1992], for example), for the latter, we construct the basis of the free algebra as a quotient of the term algebra.

Equational theories come up in semantics when non-deterministic languages are studied. They typically contain a commutative, associative, and idempotent binary operation, standing for the union of two possible branches a program may take. The associated algebras are known under the name ‘powerdomains’ and they have been the subject of detailed studies. We shall present some of their theory in the second section.

## 6.1 General techniques

### 6.1.1 Free dcpo-algebras

Let us recall the basic concepts of universal algebra so as to fix the notation for this chapter. A signature  $\Sigma = \langle \Omega, \alpha \rangle$  consists of a set  $\Omega$  of operation symbols and a map  $\alpha: \Omega \rightarrow \mathbb{N}$ , assigning to each operation symbol a (finite) arity. A  $\Sigma$ -algebra  $\underline{A} = \langle A, I \rangle$  is given by a carrier set  $A$  and an interpretation  $I$  of the operation symbols, in the sense that for  $f \in \Omega$ ,  $I(f)$  is a map from  $A^{\alpha(f)}$  to  $A$ . We also write  $f_A$  or even  $f$  for the interpreted operation symbol and speak of the operation  $f$  on  $A$ . A homomorphism between two  $\Sigma$ -algebras  $\underline{A}$  and  $\underline{B}$  is a map  $h: A \rightarrow B$  which commutes with the operations:

$$\forall f \in \Omega. h(f_A(a_1, \dots, a_{\alpha(f)})) = f_B(h(a_1), \dots, h(a_{\alpha(f)}))$$

We denote the term algebra over a set  $X$  with respect to a signature  $\Sigma$  by  $T_\Sigma(X)$ . It has the universal property that each map from  $X$  to  $A$ , where  $\underline{A} = \langle A, I \rangle$  is a  $\Sigma$ -algebra, can be extended uniquely to a homomorphism  $\bar{h}: T_\Sigma(X) \rightarrow \underline{A}$ . Let  $V$  be a fixed countable set whose elements we refer to as ‘variables’. Pairs of elements of  $T_\Sigma(V)$  are used to encode equations. An equation  $\tau_1 = \tau_2$  is said to hold in an algebra  $\underline{A} = \langle A, I \rangle$  if for each

map  $h: V \rightarrow A$  we have  $\bar{h}(\tau_1) = \bar{h}(\tau_2)$ . The pair  $\langle \bar{h}(\tau_1), \bar{h}(\tau_2) \rangle$  is also called an instance of the equation  $\tau_1 = \tau_2$ . The class of  $\Sigma$ -algebras in which each equation from a set  $\mathcal{E} \subseteq T_\Sigma(V) \times T_\Sigma(V)$  holds, is denoted by  $\mathbf{Set}(\Sigma, \mathcal{E})$ .

Here we are interested in *dcpo-algebras*, characterized by the property that the carrier set is equipped with an order relation such that it becomes a dcpo, and such that each operation is Scott-continuous. Naturally, we also require the homomorphisms to be Scott-continuous. Because of the order we can also incorporate inequalities. So from now on we let a pair  $\langle \tau_1, \tau_2 \rangle \in \mathcal{E} \subseteq T_\Sigma(V) \times T_\Sigma(V)$  stand for the inequality  $\tau_1 \sqsubseteq \tau_2$ . We use the notation  $\mathbf{DCPO}(\Sigma, \mathcal{E})$  for the class of all dcpo-algebras over the signature  $\Sigma$  which satisfy the inequalities in  $\mathcal{E}$ . For these we have:

**Proposition 6.1.1.** *For every signature  $\Sigma$  and set  $\mathcal{E}$  of inequalities, the class  $\mathbf{DCPO}(\Sigma, \mathcal{E})$  with Scott-continuous homomorphisms forms a complete category.*

**Proof.** It is checked without difficulties that  $\mathbf{DCPO}(\Sigma, \mathcal{E})$  is closed under products and equalizers, which are both defined as in the ordinary case. ■

This proves that we have one ingredient for the adjoint functor theorem, namely, a complete category  $\mathbf{DCPO}(\Sigma, \mathcal{E})$  and a (forgetful) functor  $U: \mathbf{DCPO}(\Sigma, \mathcal{E}) \rightarrow \mathbf{DCPO}$  which preserves all limits. The other ingredient is the so-called solution set condition. For this set-up it says that each dcpo can generate only set-many non-isomorphic dcpo-algebras. This is indeed the case: Given a dcpo  $D$  and a continuous map  $i: D \rightarrow A$ , where  $A$  is the carrier set of a dcpo-algebra  $\underline{A}$ , we construct the dcpo-subalgebra of  $\underline{A}$  generated by  $i(D)$  in two stages. In the first we let  $S$  be the (ordinary) subalgebra of  $\underline{A}$  which is generated by  $i(D)$ . Its cardinality is bounded by an expression depending on the cardinality of  $D$  and  $\Omega$ . Then we add to  $S$  all suprema of directed subsets until we get a sub-dcpo  $\tilde{S}$  of the dcpo  $A$ . Because we have required the operations on  $A$  to be Scott-continuous,  $\tilde{S}$  remains to be a subalgebra. The crucial step in this argument now is that the cardinality of  $\tilde{S}$  is bounded by  $2^{|S|}$  as we asked you to show in Exercise 2.3.9(33). All in all, given  $\Sigma$ , the cardinality of  $\tilde{S}$  has a bound depending on  $|D|$  and so there is only room for a set of different dcpo-algebras. Thus we have shown:

**Theorem 6.1.2.** *For every signature  $\Sigma$  and set  $\mathcal{E}$  of inequalities, the forgetful functor  $U: \mathbf{DCPO}(\Sigma, \mathcal{E}) \rightarrow \mathbf{DCPO}$  has a left adjoint.*

*Equivalently: For each dcpo  $D$  the free dcpo-algebra over  $D$  with respect to  $\Sigma$  and  $\mathcal{E}$  exists.*

The technique of this subsection is quite robust and has been used in [Nelson, 1981] for proving the existence of free algebras under more general notions of convergence than that of directed-completeness. This, however, is not the direction we are interested in, and instead we shall now turn to continuous domains.

### 6.1.2 Free continuous domain-algebras

None of the categories of approximated dcpos, or domains, we have met so far is complete. Both infinite products and equalizers may fail to exist. Hence we cannot rely on the adjoint functor theorem. While this will result in a more technical proof, there will also be a clear advantage: we will gain explicit information about the basis of the constructed free algebra, which may help us to find alternative descriptions. In the case of dcpos, such concrete representations are quite complicated, see [Nelson, 1981; Adámek *et al.*, 1982].

We denote the category of dcpo-algebras, whose carriers form a continuous domain, by  $\mathbf{CONT}(\Sigma, \mathcal{E})$  and speak of (continuous) *domain-algebras*. Again there is the obvious forgetful functor  $U: \mathbf{CONT}(\Sigma, \mathcal{E}) \rightarrow \mathbf{CONT}$ . To keep the notation manageable we shall try to suppress mention of  $U$ , in particular, we will write  $A$  for  $U(\underline{A})$  on objects and make no distinction between  $h$  and  $U(h)$  on morphisms. Let us write down the condition for adjointness on which we will base our proof:

$$\begin{array}{ccc}
 D & \xrightarrow{\eta} & F(D) \\
 & \searrow g & \downarrow \text{ext}(g) \\
 & & A
 \end{array}
 \quad
 \begin{array}{ccc}
 & & \underline{F(D)} \\
 & & \downarrow \exists! \text{ext}(g) \\
 & & \underline{A}
 \end{array}$$

$\mathbf{CONT}$ 

 $\mathbf{CONT}(\Sigma, \mathcal{E})$

In words: Suppose a signature  $\Sigma$  and a set  $\mathcal{E}$  of inequalities has been fixed. Then given a continuous domain  $D$  we must construct a dcpo-algebra  $\underline{F(D)}$ , whose carrier set  $F(D)$  is a continuous domain, and a Scott-continuous function  $\eta: D \rightarrow F(D)$  such that  $\underline{F(D)}$  satisfies the inequalities in  $\mathcal{E}$  and such that given any such domain-algebra  $\underline{A}$  and Scott-continuous map  $g: D \rightarrow A$  there is a unique Scott-continuous homomorphism  $\text{ext}(g): \underline{F(D)} \rightarrow \underline{A}$  for which  $\text{ext}(g) \circ \eta = g$ . (It may be instructive to compare this with Definition 3.1.9.)

The idea for solving this problem is to work explicitly with bases (cf. Section 2.2.6). So assume that we have fixed a basis  $\langle B, \ll \rangle$  for the continuous domain  $D$ . We will construct an abstract basis  $\langle FB, \prec \rangle$  for the desired free domain-algebra  $\underline{F(D)}$ . The underlying set  $FB$  is given by the set  $T_\Sigma(B)$  of all terms over  $B$ . On  $FB$  we have two natural order relations. The first, which we denote by  $\sqsubseteq$ , is induced by the defining set  $\mathcal{E}$  of inequalities. We can give a precise definition in the form of a deduction scheme:

Axioms:

(A1)  $t \sqsubseteq t$  for all  $t \in FB$ .

(A2)  $s \sqsubseteq t$  if this is an instance of an inequality from  $\mathcal{E}$ .

Rules:

- (R1) If  $f \in \Omega$  is an  $n$ -ary function symbol and if  $s_1 \sqsubseteq t_1, \dots, s_n \sqsubseteq t_n$  then  $f(s_1, \dots, s_n) \sqsubseteq f(t_1, \dots, t_n)$ .
- (R2) If  $s \sqsubseteq t$  and  $t \sqsubseteq u$  then  $s \sqsubseteq u$ .

The relation  $\sqsubseteq$  is the ‘least substitutive preorder’ in the terminology of [Stoughton, 1988]. It is the obvious generalization of the concept of a congruence relation to the preordered case, and indeed,  $\langle FB, \sqsubseteq \rangle$  is the free preordered algebra over  $B$ . The associated equivalence relation we denote by  $\approx$ . The factor set  $FB/\approx$  is ordered by  $\sqsubseteq$  and this is the free ordered algebra over  $B$ .

Let us now turn to the second relation on  $FB$ , namely, the one which arises from the order of approximation on  $B$ . We set  $t \prec^s t'$  if  $t$  and  $t'$  have the same structure and corresponding constants are related by  $\ll$ . Formally,  $\prec^s$  is given through the deduction scheme:

Axioms:

- (A)  $a \prec^s b$  if  $a \ll b$  in  $B$ .

Rules:

- (R) If  $f \in \Omega$  is an  $n$ -ary function symbol and if  $s_1 \prec^s t_1, \dots, s_n \prec^s t_n$  then  $f(s_1, \dots, s_n) \prec^s f(t_1, \dots, t_n)$ .

Our first observation is that  $\prec^s$  satisfies the interpolation axiom:

**Proposition 6.1.3.**  $\langle FB, \prec^s \rangle$  is an abstract basis.

**Proof.** Since  $\prec^s$  relates only terms of the same structure, it is quite obvious that it is a transitive relation. For the interpolation axiom assume that  $s \prec^s t$  holds for all elements  $s$  of a finite set  $M \subseteq FB$ . For each occurrence of a constant  $a$  in  $t$  let  $M_a$  be the set of constants which occur in the same location in one of the terms  $s \in M$ . Since  $M_a$  is finite and since  $M_a \ll a$  holds by the definition of  $\prec^s$ , we find interpolating elements  $a'$  between  $M_a$  and  $a$ . Let  $t'$  be the term in which all constants are replaced by the corresponding interpolating element. This is a term which interpolates between  $M$  and  $t$  in the relation  $\prec^s$ . ■

The question now is how to combine  $\sqsubseteq$  and  $\prec^s$ . As a guideline we take Proposition 2.2.2(2). If the inequalities tell us that  $t_1$  should be below  $s_1$  and  $s_2$  should be below  $t_2$  and if  $s_1$  approximates  $s_2$  then it should be the case that  $t_1$  approximates  $t_2$ . Hence we define  $\prec$ , the order of approximation on  $FB$ , to be the transitive closure of  $\sqsubseteq \circ \prec^s \circ \sqsubseteq$ . The following, somewhat technical properties will be instrumental for the free algebra theorem:

**Proposition 6.1.4.**

1.  $\prec^s \circ \sqsubseteq$  is contained in  $\prec^s \circ \sqsubseteq \circ \prec^s$ .
2. For every  $n \leq m \in \mathbb{N}$  we have  $(\sqsubseteq \circ \prec^s \circ \sqsubseteq)^n \subseteq (\sqsubseteq \circ \prec^s \circ \sqsubseteq)^m$ .



**Proof.** (1) Assume  $s \prec^s t \sqsubseteq u$ . Let  $C \subseteq B$  be the set of all constants which appear in the derivation of  $t \sqsubseteq u$ . For each  $c \in C$  let  $M_c$  be the set of constants which appear in  $s$  at the same place as  $c$  appears in  $t$ . Of course,  $c$  may not occur in  $t$  at all; in this case  $M_c$  will be empty. If it occurs several times then  $M_c$  can contain more than one element. In any case,  $M_c$  is finite and  $M_c \ll c$  holds. Let  $c'$  be an interpolating element between  $M_c$  and  $c$ . We now replace each constant  $c$  in the derivation of  $t \sqsubseteq u$  by the corresponding constant  $c'$  and we get a valid derivation of a formula  $t' \sqsubseteq u'$ . (The catch is that an instance of an inequality is transformed into an instance of the same inequality.) It is immediate from the construction that  $s \prec^s t' \sqsubseteq u' \prec^s u$  holds.

(2) Using (1) and the reflexivity of  $\sqsubseteq$  we get

$$\sqsubseteq \circ \prec^s \circ \sqsubseteq \subseteq \sqsubseteq \circ (\prec^s \circ \sqsubseteq \circ \prec^s) \subseteq \sqsubseteq \circ \prec^s \circ \sqsubseteq \circ \sqsubseteq \circ \prec^s \circ \sqsubseteq.$$

The general case follows by induction. ■

**Lemma 6.1.5.**  $\langle FB, \prec \rangle$  is an abstract basis.

**Proof.** Transitivity has been built in, so it remains to look at the interpolation axiom. Let  $M \prec t$  for a finite set  $M$ . From the definition of  $\prec$  we get for each  $s \in M$  a sequence of terms  $s \sqsubseteq s^1 \prec^s s^2 \sqsubseteq \dots \sqsubseteq s^{n(s)-1} \prec^s s^{n(s)} \sqsubseteq t$ . The last two steps may be replaced by  $s^{n(s)-1} \prec^s s' \sqsubseteq s'' \prec^s t$  as we have shown in the preceding proposition. The collection of all  $s''$  is finite and we find an interpolating term  $t'$  between it and  $t$  according to Proposition 6.1.3. Because of the reflexivity of  $\sqsubseteq$  we have  $M \prec t' \prec t$ . ■

So we can take as the carrier set of our free algebra over  $D$  the ideal completion of  $\langle FB, \prec \rangle$  and from Proposition 2.2.22 we know that this is a continuous domain. The techniques of Section 2.2.6 also help us to fill in the remaining pieces. The operations on  $F(D)$  are defined point-wise: If  $A_1, \dots, A_n$  are ideals and if  $f \in \Omega$  is an  $n$ -ary function symbol then we let  $f_{F(D)}(A_1, \dots, A_n)$  be the ideal which is generated by  $\{f(t_1, \dots, t_n) \mid t_1 \in A_1, \dots, t_n \in A_n\}$ . We need to know that this set is directed. It will follow if the operations on  $FB$  are monotone with respect to  $\prec$ . So assume we are given an operation symbol  $f \in \Omega$  and pairs  $s_1 \prec t_1, \dots, s_n \prec t_n$ . By definition, each pair translates into a sequence  $s_i \sqsubseteq s_i^1 \prec^s s_i^2 \sqsubseteq \dots \prec^s s_i^{m(i)} \sqsubseteq t_i$ . Now we use Proposition 6.1.4(2) to extend all these sequences to the same length  $m$ . Then we can apply  $f$  step by step, using Rules (R1) and (R) alternately:

$$\begin{aligned} f(s_1, \dots, s_n) \sqsubseteq f(s_1^1, \dots, s_n^1) &\prec^s f(s_1^2, \dots, s_n^2) \sqsubseteq \dots \\ \dots \prec^s f(s_1^m, \dots, s_n^m) &\sqsubseteq f(t_1, \dots, t_n). \end{aligned}$$

Using the remark following Proposition 2.2.24 we infer that the operations  $f_{F(D)}$  defined this way are Scott-continuous functions. Thus  $\underline{F(D)}$  is a



continuous domain-algebra. The generating domain  $D$  embeds into  $F(D)$  via the extension  $\eta$  of the monotone inclusion of  $B$  into  $FB$ .

**Theorem 6.1.6.**  *$\underline{F(D)}$  is the free continuous domain-algebra over  $D$  with respect to  $\Sigma$  and  $\mathcal{E}$ .*

**Proof.** We must show that  $\underline{F(D)}$  satisfies the inequalities in  $\mathcal{E}$  and that it has the universal property.

For the inequalities let  $\langle \tau_1, \tau_2 \rangle \in \mathcal{E}$  and let  $h: V \rightarrow F(D)$  be a map. It assigns to each variable an ideal in  $FB$ . We must show that  $\bar{h}(\tau_1)$  is a subset of  $\bar{h}(\tau_2)$ . As we have just seen, the ideal  $\bar{h}(\tau_1)$  is generated by terms of the form  $\bar{k}(\tau_1)$  where  $k$  is a map from  $V$  to  $FB$ , such that for each variable  $x \in V$ ,  $k(x) \in h(x)$ . So suppose  $s \prec \bar{k}(\tau_1)$  for such a  $k$ . Then  $\bar{k}(\tau_1) \sqsubseteq \bar{k}(\tau_2)$  is an instance of the inequality in the term algebra  $\underline{FB} = T_\Sigma(B)$  and so we know that  $s \prec \bar{k}(\tau_2)$  also holds. The term  $\bar{k}(\tau_2)$  belongs to  $\bar{h}(\tau_2)$ , again because the operations on  $\underline{F(D)}$  are defined pointwise. So  $s \in \bar{h}(\tau_2)$  as desired.

For the universal property assume that we are given a continuous map  $g: D \rightarrow A$  for a dcpo-algebra  $A$  which satisfies the inequalities from  $\mathcal{E}$ . The restriction of  $g$  to the set  $B \subseteq D$  has a unique monotone extension  $\bar{g}$  to the preordered algebra  $\langle FB, \sqsubseteq \rangle$ . We want to show that  $\bar{g}$  also preserves  $\prec^s$ . For an axiom  $a \prec^s b$  this is clear because  $g$  is monotone on  $\langle B, \ll \rangle$ . For the rules (R) we use that  $\bar{g}$  is a homomorphism and that the operations on  $A$  are monotone:

$$\begin{aligned} \bar{g}(f(s_1, \dots, s_n)) &= f_A(\bar{g}(s_1), \dots, \bar{g}(s_n)) \\ &\sqsubseteq f_A(\bar{g}(t_1), \dots, \bar{g}(t_n)) \\ &= \bar{g}(f(t_1, \dots, t_n)). \end{aligned}$$

Together this says that  $\bar{g}$  preserves the order of approximation  $\prec$  on  $FB$  and therefore it can be extended to a homomorphism  $\text{ext}(g)$  on the ideal completion  $F(D)$ . Uniqueness of  $\text{ext}(g)$  is obvious. What we have to show is that  $\text{ext}(g)$ , when restricted to  $B$ , equals  $g$ , because Proposition 2.2.24 does not give an extension but only a best approximation. We can nevertheless prove it here because  $g$  arose as the restriction of a continuous map on  $D$ . An element  $d$  of  $D$  is represented in  $F(D)$  as the ideal  $\eta(d)$  containing at least all of  $B_d = B \cap \downarrow d$  because of the axioms of our second deductive system. So we have:  $\text{ext}(g)(\eta(d)) = \bigsqcup^\uparrow \bar{g}(\eta(d)) \supseteq \bigsqcup^\uparrow \bar{g}(B_d) = \bigsqcup^\uparrow g(B_d) = g(d)$ . ■

**Theorem 6.1.7.** *For any signature  $\Sigma$  and set  $\mathcal{E}$  of inequalities the forgetful functor  $U: \mathbf{CONT}(\Sigma, \mathcal{E}) \rightarrow \mathbf{CONT}$  has a left adjoint  $F$ . It is equivalent to the restriction and corestriction of the left adjoint from Theorem 6.1.2 to  $\mathbf{CONT}$  and  $\mathbf{CONT}(\Sigma, \mathcal{E})$ , respectively.*

*In other words: Free continuous domain-algebras exist and they are also free with respect to dcpo-algebras.*

The action of the left adjoint functor on morphisms is obtained by assigning to a continuous function  $g: D \rightarrow E$  the homomorphism which extends  $\eta_E \circ g$ .

$$\begin{array}{ccc} D & \xrightarrow{\eta_D} & F(D) \\ \downarrow g & & \downarrow F(g) \\ E & \xrightarrow{\eta_E} & F(E) \end{array}$$

We want to show that  $F$  is locally continuous (Definition 5.2.3). To this end let us first look at the passage from maps to their extension.

**Proposition 6.1.8.** *The assignment  $g \mapsto \text{ext}(g)$ , as a map from  $[D \rightarrow A]$  to  $[F(D) \rightarrow A]$ , is Scott-continuous.*

**Proof.** By Proposition 2.2.25 it is sufficient to show this for the restriction of  $g$  to the basis  $B$  of  $D$ . Let  $G$  be a directed collection of monotone maps from  $B$  to  $A$  and let  $t \in FB$  be a term in which the constants  $a_1, \dots, a_n \in B$  occur. We calculate

$$\begin{aligned} \overline{\bigsqcup^\dagger G(t)} &= t[\bigsqcup^\dagger G(a_1)/a_1, \dots, \bigsqcup^\dagger G(a_n)/a_n] \\ &= \bigsqcup_{g \in G}^\dagger t[g(a_1)/a_1, \dots, g(a_n)/a_n] \\ &= \bigsqcup_{g \in G}^\dagger \bar{g}(t), \end{aligned}$$

where we have written  $t[b_1/a_1, \dots, b_n/a_n]$  for the term in which each occurrence of  $a_i$  is replaced by  $b_i$ . Restriction followed by homomorphic extension followed by extension to the ideal completion gives a sequence of continuous functions  $[D \rightarrow A] \rightarrow [B \xrightarrow{m} A] \rightarrow [FB \xrightarrow{m} A] \rightarrow [F(D) \rightarrow A]$  which equals  $\text{ext}$ .  $\blacksquare$

Cartesian closed categories can be viewed as categories in which the Hom-functor can be internalized. The preceding proposition formulates a similar closure property of the free construction: if the free construction can be cut down to a cartesian closed category then there the associated monad and the natural transformations that come with it can be internalized. This concept was introduced by Anders Kock [1970; 1972]. It has recently found much interest under the name ‘computational monads’ through the work of Eugenio Moggi [1991].

**Theorem 6.1.9.** *For any signature  $\Sigma$  and set  $\mathcal{E}$  of inequalities the composition  $U \circ F$  is a locally continuous functor on **CONT**.*

**Proof.** The action of  $U \circ F$  on morphisms is the combination of composition with  $\eta_E$  and  $\text{ext}$ . ■

If  $e: D \rightarrow E$  is an embedding then we can describe the action of  $F$ , respectively  $U \circ F$ , quite concretely. A basis element of  $F(D)$  is the equivalence class of some term  $s$ . Its image under  $F(e)$  is the equivalence class of the term  $s'$ , which we get from  $s$  by replacing all constants in  $s$  by their image under  $e$ .

If we start out with an algebraic domain  $D$  then we can choose as its basis  $K(D)$ , the set of compact elements. The order of approximation on  $K(D)$  is the order relation inherited from  $D$ , in particular, it is reflexive. From this it follows that the constructed order of approximation  $\prec$  on  $FB$  is also reflexive, whence the ideal completion of  $\langle FB, \prec \rangle$  is an algebraic domain. This gives us:

**Theorem 6.1.10.** *For any signature  $\Sigma$  and set  $\mathcal{E}$  of inequalities the forgetful functor from  $\mathbf{ALG}(\Sigma, \mathcal{E})$  to **ALG** has a left adjoint.*

Finally, let us look at  $\eta$ , which maps the generating domain  $D$  into the free algebra, and let us study the question of when it is injective. What we can say is that if injectivity fails then it fails completely:

**Proposition 6.1.11.** *For any inequational theory the canonical map  $\eta$  from a dcpo  $D$  into the free algebra  $F(D)$  over  $D$  is order-reflecting if and only if there exists a dcpo-algebra  $\underline{A}$  for this theory for which the carrier dcpo  $A$  is non-trivially ordered.*

**Proof.** Assume that there exists a dcpo-algebra  $\underline{A}$  which contains two elements  $a \sqsubset b$ . Let  $D$  be any dcpo and  $x \not\sqsubseteq y$  two distinct elements. We can define a continuous map  $g$  from  $D$  to  $A$ , separating  $x$  from  $y$  by setting

$$g(d) = \begin{cases} a, & \text{if } d \sqsubseteq y; \\ b, & \text{otherwise.} \end{cases}$$

Since  $g$  equals  $\text{ext}(g) \circ \eta$ , where  $\text{ext}(g)$  is the unique homomorphism from  $F(D)$  to  $\underline{A}$ , it cannot be that  $\eta(x) \sqsubseteq \eta(y)$  holds.

The converse is trivial, because  $\eta$  must be monotone. ■

### 6.1.3 Least elements and strict algebras

We have come across strict functions several times already. It therefore seems worthwhile to study the problem of free algebras also in this context. But what should a strict algebra be? There are several possibilities as to what to require of the operations on such an algebra:

1. An operation which is applied to arguments, one of which is bottom, returns bottom.

2. An operation applied to the constantly bottom vector returns bottom.
3. An operation of arity greater than 0 applied to the constantly bottom vector returns bottom.

Luckily, we can leave this open as we shall see shortly. All we need is:

**Definition 6.1.12.** A *strict dcpo-algebra* is a dcpo-algebra for which the carrier set contains a least element. A *strict homomorphism* between strict algebras is a Scott-continuous homomorphism which preserves the least element.

For pointed dcpos the existence of free strict dcpo-algebras can be established as before through the adjoint functor theorem. For pointed domains the construction of the previous subsection can be adapted by adding a further axiom to the first deduction scheme:

(A3)  $\perp \sqsubseteq t$  for all  $t \in FB$ .

Thus we have:

**Theorem 6.1.13.** *Free strict dcpo- and domain-algebras exist, that is, the forgetful functors*

$$\begin{aligned} \mathbf{DCPO}_{\perp!}(\Sigma, \mathcal{E}) &\longrightarrow \mathbf{DCPO}_{\perp!}, \\ \mathbf{CONT}_{\perp!}(\Sigma, \mathcal{E}) &\longrightarrow \mathbf{CONT}_{\perp!}, \\ \text{and } \mathbf{ALG}_{\perp!}(\Sigma, \mathcal{E}) &\longrightarrow \mathbf{ALG}_{\perp!} \end{aligned}$$

*have left adjoints.*

Let us return to the problem of strict operations. The solution is that we can add a nullary operation 0 to the signature and the inequality  $0 \sqsubseteq x$  to  $\mathcal{E}$  without changing the free algebras. Because of axiom (A3) we have  $\perp \sqsubseteq 0$  and because of the new inequality we have  $0 \sqsubseteq \perp$ . Therefore the new operation must be interpreted by the bottom element. The advantage of having bottom explicitly in the signature is that we can now formulate equations about strictness of operations. For example, the first possibility mentioned at the beginning can be enforced by adding to  $\mathcal{E}$  the inequality

$$f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_{\alpha(f)}) \sqsubseteq 0$$

for all operation symbols  $f$  of positive arity and all  $1 \leq i \leq \alpha(f)$ . The corresponding free algebras then exist by the general theorem.

More problematic is the situation with  $\mathbf{DCPO}_{\perp}$  (respectively  $\mathbf{CONT}_{\perp}$  and  $\mathbf{ALG}_{\perp}$ ). The existence of a least element in the generating dcpo does not imply the existence of a least element in the free algebra (Exercise 6.2.23(2)). Without it, we cannot make use of local continuity in domain equations. Furthermore, even if the free algebra has a least element, it need not be the case that  $\eta$  is strict (Exercise 6.2.23(3)). The

same phenomenon appears if we restrict attention to any of the cartesian closed categories exhibited in Chapter 4. The reason is that we require a special structure of the objects of our category but allow morphisms which do not preserve this structure. It is therefore always an interesting fact if the general construction for a particular algebraic theory can be restricted and corestricted to one of these sub-categories. In the case that the general construction does not yield the right objects it may be that a different construction is needed. This has been tried for the Plotkin powerdomain in several attempts by Karel Hrbacek, [1987; 1989; 1988] but a satisfactory solution was obtained only at the cost of changing the morphisms between continuous algebras.

On a more positive note, we can say:

**Proposition 6.1.14.** *If the free functor maps finite pointed posets to finite pointed posets then it restricts and corestricts to bifinite domains.*

## 6.2 Powerdomains

### 6.2.1 The convex or Plotkin powerdomain

**Definition 6.2.1.** The *convex* or *Plotkin powertheory* is defined by a signature with one binary operation  $\sqcup$  and the equations

1.  $x \sqcup y = y \sqcup x$  (commutativity)
2.  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$  (associativity)
3.  $x \sqcup x = x$  (idempotence)

The operation  $\sqcup$  is called *formal union*.

A dcpo-algebra with respect to this theory is called a *dcpo-semilattice*. The free dcpo-semilattice over a dcpo  $D$  is called the *Plotkin powerdomain* of  $D$  and it is denoted by  $P^P(D)$ .

Every semilattice can be equipped with an order by setting

$$x \leq y \text{ if } x \sqcup y = y.$$

Formal union then becomes the join in the resulting ordered set. On a dcpo-semilattice this order has little to do with the domain ordering and it is not in the focus of our interest.

The free semilattice over a set  $X$  is given by the set of all non-empty finite subsets of  $X$ , where formal union is interpreted as actual union of sets. This gives us the first half of an alternative description of the Plotkin powerdomain over a continuous domain  $D$  with basis  $B$ . Its basis  $FB$ , which we constructed as the term algebra over  $B$ , is partitioned into equivalence classes by  $\approx$ , the equivalence relation derived from  $\sqsubseteq$ , that is, from the defining equations. These equivalence classes are in one-to-one correspondence with finite subsets of  $B$ . Indeed, given a term from  $FB$ , we can re-arrange it because of associativity and commutativity, and because of idempotence we can make sure that each constant occurs just once.



Remember that we have set up the order of approximation  $\prec$  on  $FB$  as the transitive closure of  $\sqsubseteq \circ \prec^s \circ \sqsubseteq$ . This way we have ensured that an ideal in  $FB$  contains only full equivalence classes with respect to  $\approx$ . We may therefore replace  $FB$  by  $\mathfrak{P}_f(B)$ , the set of finite non-empty subsets of  $B$ , where we associate with a term  $t \in FB$  the set  $[t]$  of constants appearing in  $t$ .

Let us now also transfer the order of approximation to the new basis.

**Definition 6.2.2.** Two subsets  $M$  and  $N$  of a set equipped with a relation  $R$  are in the *Egli–Milner relation*, written as  $M R_{EM} N$ , if the following two conditions are satisfied:

$$\forall a \in M \exists b \in N. a R b$$

$$\forall b \in N \exists a \in M. a R b.$$

Here we are talking about finite subsets of  $\langle B, \ll \rangle$ , so we write  $\ll_{EM}$  for the Egli–Milner relation between finite subsets of  $B$ . Let us establish the connection between  $\ll_{EM}$  on  $\mathfrak{P}_f(B)$  and  $\prec$  on  $FB$ . Firstly, if  $s \prec^s t$  then by definition each constant in  $t$  is matched by a constant in  $s$  which approximates it and vice versa. These are just the conditions for  $[s] \ll_{EM} [t]$ . Since  $\ll_{EM}$  is transitive, we find that  $s \prec t$  implies  $[s] \ll_{EM} [t]$  in general. Conversely, if two finite subsets  $M = \{a_1, \dots, a_m\}$  and  $N = \{b_1, \dots, b_n\}$  of  $B$  are related by  $\ll_{EM}$  then we can build terms  $s$  and  $t$ , such that  $[s] = M$ ,  $[t] = N$ , and  $s \prec^s t$  hold. This is done as follows. For each  $a_i \in M$  let  $b_{j(i)}$  be an element of  $N$  such that  $a_i \ll b_{j(i)}$  and for each  $b_j \in N$  let  $a_{i(j)}$  be an element of  $M$  such that  $a_{i(j)} \ll b_j$ . Then we can set

$$s = (a_1 \cup \dots \cup a_m) \cup (a_{i(1)} \cup \dots \cup a_{i(n)})$$

and  $t = (b_{j(1)} \cup \dots \cup b_{j(m)}) \cup (b_1 \cup \dots \cup b_n).$

We have proved:

**Theorem 6.2.3.** *The Plotkin powerdomain of a continuous domain  $D$  with basis  $\langle B, \ll \rangle$  is given by the ideal completion of  $(\mathfrak{P}_f(B), \ll_{EM})$ .*

An immediate consequence of this characterization is that the Plotkin powerdomain of a finite pointed poset is again finite and pointed. By Proposition 6.1.14, the Plotkin powerdomain of a bifinite domain is again bifinite. This is almost the best result we can obtain. The Plotkin power construction certainly destroys all properties of being lattice-like, see Exercise 6.2.23(8). It is, on the other hand, not completely haphazard, in the sense that not every finite poset is a sub-domain of a powerdomain of some other poset. This was shown in [Nüßler, 1992].

The passage from terms to finite sets has reduced the size of the basis for the powerdomain drastically. Yet, it is still possible to get an even leaner

representation. We present this for algebraic domains only. For continuous domains a similar treatment is possible but it is less intuitive. Remember that abstract bases for algebraic domains are preordered sets.

**Definition 6.2.4.** For a subset  $M$  of a preordered set  $\langle B, \sqsubseteq \rangle$  let the *convex hull*  $Cx(M)$  be defined by

$$\{a \in B \mid \exists m, n \in M. m \sqsubseteq a \sqsubseteq n\}.$$

A set which coincides with its convex hull is called *convex*.

The following properties are easily checked:

**Proposition 6.2.5.** Let  $\langle B, \sqsubseteq \rangle$  be a preordered set and  $M, N$  be subsets of  $B$ .

1.  $Cx(M) = \uparrow M \cap \downarrow M$ .
2.  $M \subseteq Cx(M)$ .
3.  $Cx(Cx(M)) = Cx(M)$ .
4.  $M \subseteq N \implies Cx(M) \subseteq Cx(N)$ .
5.  $M =_{EM} Cx(M)$ .
6.  $M =_{EM} N$  if and only if  $Cx(M) = Cx(N)$ .

While  $\langle \mathfrak{P}_f(K(D)), \sqsubseteq_{EM} \rangle$  is only a preordered set, parts (5) and (6) of the preceding proposition suggests how to replace it with an ordered set. Writing  $\mathfrak{P}_{Cx,f}(K(D))$  for the set of finitely generated convex subsets of  $K(D)$ , we have:

**Proposition 6.2.6.** The Plotkin powerdomain of an algebraic domain  $D$  is isomorphic to the ideal completion of  $\langle \mathfrak{P}_{Cx,f}(K(D)), \sqsubseteq_{EM} \rangle$ .

This explains the alternative terminology ‘convex powerdomain’. We will sharpen this description in Section 6.2.3.

For examples of how the Plotkin powerdomain can be used in semantics, we refer to [Hennessy and Plotkin, 1979; Abramsky, 1991a].

### 6.2.2 One-sided powerdomains

**Definition 6.2.7.** If the Plotkin powertheory is augmented by the inequality

$$x \sqsubseteq x \cup y$$

then we obtain the *Hoare* or *lower powertheory*. Algebras for this theory are called *inflationary semilattices*. The free inflationary semilattice over a dcpo  $D$  is called the *lower* or *Hoare powerdomain* of  $D$ , and it is denoted by  $P^H(D)$ .

Similarly, the terminology concerning the inequality

$$x \sqsupseteq x \cup y$$

is *upper* or *Smyth powerdomain*, *deflationary semilattice*, and  $P^S(D)$ .

It is a consequence of the new inequality that the semilattice ordering and the domain ordering coincide in the case of the Hoare powertheory. For the Smyth powertheory the semilattice ordering is the reverse of the domain ordering. This forces these powerdomains to have additional structure.

**Proposition 6.2.8.**

1. *The Hoare powerdomain of any dcpo is a lattice which has all non-empty suprema and bounded infima. The sup operation is given by formal union.*
2. *The Smyth powerdomain of any dcpo has binary infima. They are given by formal union.*

Unfortunately, the existence of binary infima does not force a domain into one of the cartesian closed categories of Chapter 4. We take up this question again in the next subsection.

Let us also study the bases of these powerdomains as derived from a given basis  $\langle B, \ll \rangle$  of a continuous domain  $D$ . The development proceeds along the same lines as for the Plotkin powertheory. The equivalence relation induced by the equations and the new inequality has not changed, so we may again replace  $FB$  by the set  $\mathfrak{P}_f(B)$  of finite subsets of  $B$ . The difference is wholly in the associated preorder on  $\mathfrak{P}_f(B)$ .

**Proposition 6.2.9.** *For  $M$  and  $N$  finite subsets of a basis  $\langle B, \ll \rangle$  we have*

$$M \sqsubseteq N \text{ if and only if } M \subseteq \downarrow N$$

*in the case of the Hoare powertheory and*

$$M \sqsubseteq N \text{ if and only if } N \subseteq \uparrow M$$

*for the Smyth powertheory.*

The restricted order of approximation  $\prec^s$  is as before given by the Egli–Milner relation  $\ll_{EM}$ . As prescribed by the general theory we must combine it with inclusion (for the lower theory) and with reversed inclusion (for the upper theory), respectively. Without difficulties one obtains the following connection:

$$s \prec_H t \text{ if and only if } \forall a \in [s] \exists b \in [t]. a \ll b$$

and

$$s \prec_S t \text{ if and only if } \forall b \in [t] \exists a \in [s]. a \ll b.$$

So each of the one-sided theories is characterized by one half of the Egli–Milner ordering. Writing  $\ll_H$  and  $\ll_S$  for these we can formulate

**Theorem 6.2.10.** *Let  $D$  be a continuous domain with basis  $\langle B, \ll \rangle$ .*

1. The Hoare powerdomain of  $D$  is isomorphic to the ideal completion of  $\langle \mathfrak{P}_f(B), \ll_H \rangle$ .
2. The Smyth powerdomain of  $D$  is isomorphic to the ideal completion of  $\langle \mathfrak{P}_f(B), \ll_S \rangle$ .

For algebraic domains we can replace the preorders on  $\mathfrak{P}_f(B)$  by an ordered set in both cases.

**Proposition 6.2.11.** *For subsets  $M$  and  $N$  of a preordered set  $\langle B, \leq \rangle$  we have*

1.  $M =_H \downarrow M$ ,
2.  $M \leq_H N$  if and only if  $\downarrow M \subseteq \downarrow N$ ,

and

3.  $M =_S \uparrow M$ ,
4.  $M \leq_S N$  if and only if  $\uparrow M \supseteq \uparrow N$ .

Writing  $\mathfrak{P}_{L,f}(B)$  for the set of finitely generated lower subsets of  $B$  and  $\mathfrak{P}_{U,f}(B)$  for the set of finitely generated upper subsets of  $B$ , we have

**Proposition 6.2.12.** *Let  $D$  be an algebraic domain.*

1. The Hoare powerdomain  $P^H(D)$  of  $D$  is isomorphic to the ideal completion of  $\langle \mathfrak{P}_{L,f}(K(D)), \subseteq \rangle$ .
2. The Smyth powerdomain  $P^S(D)$  of  $D$  is isomorphic to the ideal completion of  $\langle \mathfrak{P}_{U,f}(K(D)), \supseteq \rangle$ .

From this description we can infer through Proposition 6.1.14 that the Smyth powerdomain of a bifinite domain is again bifinite. Since a deflationary semilattice has binary infima anyway, we conclude that the Smyth powerdomain of a bifinite domain is actually a bc-domain. For a more general statement see Corollary 6.2.15.

### 6.2.3 Topological representation theorems

The objective of this subsection is to describe the powerdomains we have seen so far directly as spaces of certain subsets of the given domain, without recourse to bases and the ideal completion. It will turn out that the characterizations of Proposition 6.2.6 and Proposition 6.2.12 can be extended nicely once we allow ourselves topological methods.

**Theorem 6.2.13.** *The Hoare powerdomain of a continuous domain  $D$  is isomorphic to the lattice of all non-empty Scott-closed subsets of  $D$ . Formal union is interpreted by actual union.*

**Proof.** Let  $\langle B, \ll \rangle$  be a basis for  $D$ . We establish an isomorphism with the representation of Theorem 6.2.10. Given an ideal  $I$  of finite sets in  $P^H(D)$  we map it to  $\phi^H(I) = \text{Cl}(\bigcup I)$ , the Scott-closure of the union of all these sets. Conversely, for a non-empty Scott-closed set  $A$  we let

$\psi^H(A) = \mathfrak{P}_f(\downarrow A \cap B)$ , the set of finite sets of basis elements approximating some element in  $A$ . We first check that  $\psi^H(A)$  is indeed an ideal with respect to  $\ll_H$ . It is surely non-empty as  $A$  was assumed to contain elements. Given two finite subsets  $M$  and  $N$  of  $\downarrow A \cap B$  then we can apply the interpolation axiom to get finite subsets  $M'$  and  $N'$  with  $M \ll_{EM} M'$  and  $N \ll_{EM} N'$ . An upper bound for  $M$  and  $N$  with respect to  $\ll_H$  is then given by  $M' \cup N'$ . It is also clear that the Scott closure of  $\downarrow A \cap B$  gives  $A$  back again because every element of  $D$  is the directed supremum of basis elements. Hence  $\phi^H \circ \psi^H = \text{id}$ . Starting out with an ideal  $I$ , we must show that we get it back from  $\phi^H(I)$ . So let  $M \in I$ . By the roundness of  $I$  (see the discussion before Definition 2.2.21) there is another finite set  $M' \in I$  with  $M \ll_H M'$ . So for each  $a \in M$  there is  $b \in M'$  with  $a \ll b$ . Since all elements of  $I$  are contained in  $\phi^H(I)$ , we have that  $a$  belongs to  $\downarrow \phi(I) \cap B$ . Conversely, if  $a$  is an element of  $\downarrow \phi(I) \cap B$  then  $\uparrow a \cap \phi(I)$  is not empty and therefore must meet  $\bigcup I$  as  $D \setminus \uparrow a$  is closed. The set  $\{a\}$  is then below some element of  $I$  under the  $\ll_H$ -ordering. Monotonicity of the isomorphisms is trivial and the representation is proved.

Formal union applied to two ideals returns the ideal of unions of the constituting sets. Under the isomorphism this operation is transformed into union of closed subsets. ■

This theorem holds not just for continuous domains but also for all dcpos and even all  $T_0$ -spaces. See [Schalk, 1993] for this. We can also get the full complete lattice of all closed sets if we add to the Hoare powertheory a nullary operation  $e$  and the equations

$$e \sqcup x = x \sqcup e = x.$$

Alternatively, we can take the strict free algebra with respect to the Hoare powertheory. If the domain has a least element then these adjustments are not necessary, a least element for the Hoare powerdomain is  $\{\perp\}$ . Homomorphisms, however, will only preserve non-empty suprema.

The characterization of the Smyth powerdomain builds on the material laid out in Section 4.2.3. In particular, recall that a Scott-compact saturated set in a continuous domain has a Scott-open filter of open neighbourhoods and that each Scott-open filter in  $\sigma_D$  arises in this way.

**Theorem 6.2.14.** *The Smyth powerdomain of a continuous domain  $D$  is isomorphic to the set  $\kappa_D \setminus \{\emptyset\}$  of non-empty Scott-compact saturated subsets ordered by reversed inclusion. Formal union is interpreted as union.*

**Proof.** Let  $\langle B, \ll \rangle$  be a basis for  $D$ . We show that  $\kappa_D \setminus \{\emptyset\}$  is isomorphic to  $P^S(D) = \text{Idl}(\mathfrak{P}_f(B), \ll_S)$ . Given an ideal  $I$  we let  $\phi^S(I)$  be  $\bigcap_{M \in I} \uparrow M$ . This constitutes a monotone map from  $P^S(D)$  to  $\kappa_D \setminus \{\emptyset\}$  by Proposition 4.2.14. In the other direction, we assign to a compact saturated set  $A$  the set



$\psi^S(A)$  of all finite sets  $M \subseteq B$  such that  $A \subseteq \uparrow M$ . Why is this an ideal? For every open neighborhood  $O$  of  $A$  we find a finite set  $M$  of basis elements contained in  $O$  such that  $A \subseteq \uparrow M$  because  $A$  is compact and  $O = \bigcup_{b \in O \cap B} \uparrow b$  (Proposition 2.3.6). Then given two finite sets  $M$  and  $N$  in  $\psi^S(A)$  an upper bound for them is any such finite set  $P$  with  $A \subseteq \uparrow P \subseteq \uparrow M \cap \uparrow N$ . Clearly,  $\psi^S$  is monotone as  $\kappa_D \setminus \{\emptyset\}$  is equipped with reversed inclusion.

Let us show that  $\psi^S \circ \phi^S$  is the identity on  $\mathcal{P}^S(D)$ . For  $M \in I$  let  $M' \in I$  be above  $M$  in the  $\ll_S$ -ordering. Then  $\phi^S(I) \subseteq \uparrow M' \subseteq \uparrow M$  and so  $M$  belongs to  $\psi^S \circ \phi^S(I)$ . Conversely, every neighborhood of  $\phi^S(I)$  contains some  $\uparrow M$  with  $M \in I$  as we saw in Proposition 4.2.14. So if  $\phi^S(I)$  is contained in  $\uparrow N$  for some finite set  $N \subseteq B$ , then there are  $M$  and  $M'$  in  $I$  with  $M \subseteq \uparrow N$  and  $M \ll_S M'$ . Hence  $N \ll_S M'$  and  $N$  belongs to  $I$ .

The composition  $\phi^S \circ \psi^S$  is clearly the identity as we just saw that every neighborhood of a compact set contains a finitely generated one and as every saturated set is the intersection of its neighborhoods.

The claim about formal union follows because on powersets union and intersection completely distribute:  $\phi^S(I \cup J) = \bigcap_{M \in I, N \in J} \uparrow(M \cup N) = \bigcap_{M \in I, N \in J} (\uparrow M \cup \uparrow N) = \bigcap_{M \in I} \uparrow M \cup \bigcap_{N \in J} \uparrow N = \phi^S(I) \cup \phi^S(J)$ . ■

For this theorem continuity is indispensable. A characterization of the free deflationary semilattice over an arbitrary dcpo is not known. The interested reader may consult Heckmann [1990; 1993a] and [Schalk, 1993] for a discussion of this open problem.

**Corollary 6.2.15.** *The Smyth powerdomain of a coherent domain with bottom is a bc-domain.*

**Proof.** That two compact saturated sets  $A$  and  $B$  are bounded by another one,  $C$ , simply means  $C \subseteq A \cap B$ . In this case  $A \cap B$  is not empty. It is compact saturated by the very definition of coherence. ■

Let us now turn to the Plotkin powerdomain. An ideal  $I$  of finite sets ordered by  $\ll_{EM}$  will generate ideals with respect to both coarser orders  $\ll_H$  and  $\ll_S$ . We can therefore associate with  $I$  a Scott-closed set  $\phi^H(I) = \text{Cl}(\bigcup I)$  and a compact saturated set  $\phi^S(I) = \bigcap_{M \in I} \uparrow M$ . However, not every such pair arises in this way; the Plotkin powerdomain is not simply the product of the two one-sided powerdomains. We will be able to characterize them in two special cases: for countably based domains and for coherent domains. The general situation is quite hopeless, as is illustrated by Exercise 6.2.23(11). In both special cases we do want to show that  $I$  is faithfully represented by the intersection  $\phi(I) = \phi^H(I) \cap \phi^S(I)$ . In the first case we will need the following weakening of the Egli-Milner ordering:

**Definition 6.2.16.** For a dcpo  $D$  we let  $\text{Lens}(D)$  be the set of all non-empty subsets of  $D$  which arise as the intersection of a Scott-closed and

a compact saturated subset. The elements of  $\text{Lens}(D)$  we call *lenses*. On  $\text{Lens}(D)$  we define the *topological Egli-Milner ordering*,  $\sqsubseteq_{TEM}$ , by

$$K \sqsubseteq_{TEM} L \text{ if } L \subseteq \uparrow K \text{ and } K \subseteq \text{Cl}(L).$$

**Proposition 6.2.17.** *Let  $D$  be a dcpo.*

1. *Every lens is convex and Scott-compact.*
2. *A canonical representation for a lens  $L$  is given by  $\uparrow L \cap \text{Cl}(L)$ .*
3. *The topological Egli-Milner ordering is anti-symmetric on  $\text{Lens}(D)$ .*

**Proof.** Convexity is clear as every lens is the intersection of a lower and an upper set. An open covering of a lens  $L = C \cap U$ , where  $C$  is closed and  $U$  compact saturated, may be extended to a covering of  $U$  by adding the complement of  $C$  to the cover. This proves compactness. Since all Scott-open sets are upwards closed, compactness of a set  $A$  implies the compactness of  $\uparrow A$ . Using convexity, we get  $L = \uparrow L \cap \downarrow L \subseteq \uparrow L \cap \text{Cl}(L)$  and using boolean algebra we calculate  $\uparrow L = \uparrow(C \cap U) \subseteq \uparrow U = U$  and  $\text{Cl}(L) = \text{Cl}(C \cap U) \subseteq \text{Cl}(C) = C$ , so  $\uparrow L \cap \text{Cl}(L) \subseteq U \cap C = L$ . Then if  $K =_{TEM} L$  we have  $\uparrow K = \uparrow L$  and  $\text{Cl}(K) = \text{Cl}(L)$ . Equality of  $K$  and  $L$  follows. ■

Before we can prove the representation theorem we need yet another description of the lens  $\phi(I)$ .

**Lemma 6.2.18.** *Let  $D$  be a continuous domain with basis  $B$  and let  $I$  be an ideal in  $\langle \mathfrak{P}_f(B), \ll_{EM} \rangle$ . Then  $\phi(I) = \{\bigcup^{\uparrow} A \mid A \subseteq \bigcup I \text{ directed and } A \cap M \neq \emptyset \text{ for all } M \in I\}$ .*

**Proof.** The elements of the set on the right clearly belong to the Scott-closure of  $\bigcup I$ . They are also contained in  $\phi^S(I)$  because  $\bigcup^{\uparrow} A$  is above some element in  $A \cap M$  for each  $M \in I$ .

Conversely, let  $x \in \phi(I)$  and let  $a \in A = \downarrow x \cap B$ . The set  $\uparrow a$  is Scott-open and must therefore meet some  $M \in I$ . From the roundness of  $I$  we get  $M' \in I$  with  $M \ll_{EM} M'$ . The set  $M \cup \{a\}$  also approximates  $M'$  and so it is contained in  $I$ . Hence  $a \in \bigcup I$ . Furthermore, given any  $M \in I$ , let again  $M' \in I$  be such that  $M \ll_{EM} M'$ . Then  $x$  is above some element of  $M'$  as  $\phi(I) \subseteq \uparrow M'$  and therefore  $m \ll x$  holds for some  $m \in M$ . ■

**Theorem 6.2.19.** *Let  $D$  be an  $\omega$ -continuous domain. The Plotkin powerdomain  $P^P(D)$  is isomorphic to  $\langle \text{Lens}(D), \sqsubseteq_{TEM} \rangle$ . Formal union is interpreted as union followed by topological convex closure.*

**Proof.** Let  $\langle B, \ll \rangle$  be a countable basis of  $D$ . We have already defined the map  $\phi: P^P(D) \rightarrow \text{Lens}(D)$ . In the other direction we take the function  $\psi$  which assigns to a lens  $K$  the set  $\psi^H(\text{Cl}(K)) \cap \psi^S(\uparrow K)$ . (Well-definedness

of  $\psi$  is left as an exercise.) Before we can prove that these maps constitute a pair of isomorphisms, we need the following information about reconstructing  $\phi^H(I)$  and  $\phi^S(I)$  from  $\phi(I)$ .

1.  $\phi^S(I) = \uparrow\phi(I)$ : Since  $\phi^S(I)$  is an upper set which contains  $\phi(I)$ , only one inclusion can be in doubt. Let  $x \in \phi^S(I)$  and  $I' = \{M \cap \downarrow x \mid M \in I\}$ . Firstly, each set in  $I'$  is non-empty and, secondly, we have  $M \cap \downarrow x \ll_S N \cap \downarrow x$  whenever  $M \ll_{EM} N$ . Calculating  $\phi^S(I')$  in the continuous domain  $\phi^H(I)$  gives us a non-empty set which is below  $x$  and contained in the lens  $\phi(I)$ .

2.  $\phi^H(I) = \text{Cl}(\phi(I))$ : Again, only one inclusion needs an argument. We show that every element of  $\downarrow\phi^H(I) \cap B$  belongs to  $\downarrow\phi(I)$ . Given a basis element  $a$  approximating some element of  $\phi^H(I)$  then we already know that it belongs to  $\bigcup I$ . Let  $M \in I$  be some set which contains  $a$ . Using countability of the basis we may assume that  $M$  extends to a cofinal chain in  $I$  (Proposition 2.2.13):  $M = M_0 \ll_{EM} M_1 \ll_{EM} M_2 \ll_{EM} \dots$ . König's Lemma then tells us that we can find a chain of elements  $a = a_0 \ll a_1 \ll a_2 \ll \dots$  where  $a_n \in M_n$ . The supremum  $x = \bigsqcup_{n \in \mathbb{N}} a_n$  belongs to  $\phi(I)$  and is above  $a$ .

3.  $\phi$  is monotone: Let  $I \subseteq I'$  be two ideals in  $\langle \mathfrak{P}_f(B), \ll_{EM} \rangle$ . The larger ideal results in a bigger lower set  $\phi^H(I')$  and a smaller upper set  $\phi^S(I')$ . Using 1 and 2 we can calculate for the corresponding lenses:

$$\begin{aligned}\phi(I) &\subseteq \phi^H(I) \subseteq \phi^H(I') = \text{Cl}(\phi(I')), \\ \phi(I') &\subseteq \phi^S(I') \subseteq \phi^S(I) = \uparrow\phi(I).\end{aligned}$$

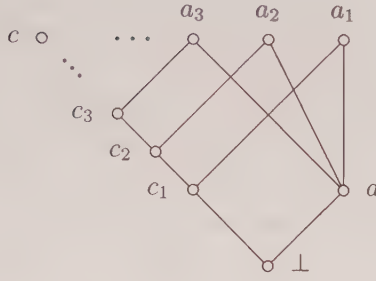
So  $\phi(I) \sqsubseteq_{TEM} \phi(I')$  as desired.

4. The monotonicity of  $\psi$  follows by construction and one half of the topological Egli Milner ordering:  $K \subseteq \uparrow M$  implies  $L \subseteq \uparrow M$  if we assume  $K \sqsubseteq_{TEM} L$ .

5.  $\phi \circ \psi = \text{id}$ : Given a lens  $L = C \cap U$  we clearly have  $\phi^S(\psi(L)) \supseteq L$ . Using the continuity of  $D$  and the compactness of  $L$  we infer that  $\phi^S(\psi(L))$  must equal  $\uparrow L$ . Every basis element approximating some element of  $L$  occurs in some set of  $\psi(L)$ , so  $\phi^H(\psi(L)) = \text{Cl}(L)$  is clear. Proposition 6.2.17 above then implies that  $\phi \circ \psi(L)$  gives back  $L$ .

6.  $\psi \circ \phi = \text{id}$ : Given an ideal  $I$  we know that each  $M \in I$  covers the lens  $\phi(I)$  in the sense of  $\uparrow M \supseteq \phi(I)$ . So  $M$  is contained in  $\psi^S(\phi(I))$ . By (2), we also have that  $M$  is contained in  $\psi^H(\text{Cl}(\phi(I)))$ . Conversely, if  $\uparrow M \supseteq \phi(I)$  for a finite set  $M$  of basis elements contained in  $\downarrow\phi(I)$ , then for some  $N \in I$  we have  $\uparrow M \supseteq N$  by the Hofmann–Mislove Theorem 4.2.14. For this  $N$  we have  $M \ll_S N$ . On the other hand, each element  $a$  of  $M$  approximates some  $x \in \phi(I)$  and hence belongs to some  $N_a \in I$ . An upper bound for  $N$  and all  $N_a$  in  $I$ , therefore, is above  $M$  in  $\ll_{EM}$ , which shows that  $M$  must belong to  $I$ .

7. In the representation theorems for the one-sided powerdomains we have shown that formal union translates to actual union. We combine this



**Fig. 13.** An algebraic domain in which topological Egli-Milner ordering and ordinary Egli-Milner ordering do not coincide.

for the convex setting:  $\phi(I \uplus J) = \phi^H(I \uplus J) \cap \phi^S(I \uplus J) = (\phi^H(I) \cup \phi^H(J)) \cap (\phi^S(I) \cup \phi^S(J)) = (\text{Cl}(\phi(I)) \cup \text{Cl}(\phi(J))) \cap (\uparrow\phi(I) \cup \uparrow\phi(J)) = \text{Cl}(\phi(I) \cup \phi(J)) \cap \uparrow(\phi(I) \cup \phi(J))$ . ■

Note that we used countability of the basis only for showing that  $\phi^H(I)$  can be recovered from  $\phi(I)$ . In general, this is wrong. Exercise 6.2.23(11) discusses an example.

The substitution of topological closure for downward closure was also necessary, as the example in Figure 13 shows. There, the set  $A = \uparrow a$  is a lens but its downward closure is not Scott-closed,  $c$  is missing. The set  $A \cup \{c\}$  is also a lens. It is below  $A$  in the topological Egli-Milner order but not in the plain Egli-Milner order. The convex closure of the union of the two lenses  $\{\perp\}$  and  $A$  is not a lens,  $c$  must be added.

A better representation theorem is obtained if we pass to coherent domains (Section 4.2.3). (Note that the example in Figure 13 is not coherent, because the set  $\{c_1, a\}$  has infinitely many minimal upper bounds, violating the condition in Proposition 4.2.17.) We first observe that lenses are always Lawson-closed sets. If the domain is coherent then this implies that they are also Lawson-compact. Compactness will allow us to use downward closure instead of topological closure.

**Lemma 6.2.20.** *Let  $L$  be a Lawson-compact subset of a continuous domain  $D$ . Then  $\downarrow L$  is Scott-closed.*

**Proof.** Let  $x$  be an element of  $D$  which does not belong to  $\downarrow L$ . For each  $y \in L$  there exists  $b_y \ll x$  such that  $b_y \not\sqsubseteq y$ . The set  $D \setminus \uparrow b_y$  is Lawson-open and contains  $y$ . By compactness, finitely many such sets cover  $L$ . Let  $b$  be an upper bound for the associated basis elements approximating  $x$ . Then  $\uparrow b$  is an open neighbourhood of  $x$  which does not intersect  $L$ . Hence  $\downarrow L$  is closed. ■



**Corollary 6.2.21.** *The lenses of a coherent domain are precisely the convex Lawson-compact subsets. For these, topological Egli–Milner ordering and Egli–Milner ordering coincide.*

**Theorem 6.2.22.** *Let  $D$  be a coherent domain. The Plotkin powerdomain of  $D$  is isomorphic to  $\langle \text{Lens}(D), \sqsubseteq_{EM} \rangle$ . Formal union is interpreted as union followed by convex closure.*

**Proof.** The differences to the proof of Theorem 6.2.19, which are not taken care of by the preceding corollary, concern part 2. We must show that  $\text{Cl}(\phi(I)) = \downarrow\phi(I)$  contains all of  $\downarrow\phi^H(I) \cap B$ . In the presence of coherence this can be done through the Hofmann–Mislove Proposition 4.2.14. The lower set  $\phi^H(I)$  is a continuous domain in itself. For an element  $a$  of  $\downarrow\phi^H(I) \cap B$  we look at the filtered collection of upper sets  $J = \{\uparrow a \cap \uparrow M \mid M \in I\}$ . Each of these is non-empty, because  $a$  belongs to some  $M \in I$ , and compact saturated because of coherence. Hence  $\bigcap J$  is non-empty. It is also contained in  $\phi(I)$  and above  $a$ . ■

#### 6.2.4 Hyperspaces and probabilistic powerdomains

In our presentation of powerdomains we have emphasized the feature that they are free algebras with respect to certain (in-)equational theories. From the general existence theorem for such algebras we derived concrete representations as sets of subsets. This is the approach which in the realm of domain theory was suggested first by Matthew Hennessy and Gordon Plotkin in [Hennessy and Plotkin, 1979] but it has a rather long tradition in algebraic semantics (see e.g. [Nivat and Reynolds, 1985]). However, it is not the only viewpoint one can take. One may also study certain sets of subsets of domains in their own right. In topology, this study of ‘hyperspaces’, as they are called, is a long-standing tradition, starting with Felix Hausdorff [1914] and Leopold Vietoris [1921; 1922]. It is also how the subject started in semantics and, indeed, continues to be developed. A hyperspace can be interesting even if an equational characterization cannot be found or can be found only in restricted settings. Recent examples of this are the set-domains introduced by Peter Buneman [Buneman *et al.*, 1988; Gunter, 1992a; Heckmann, 1990; Puhlmann, 1993; Heckmann, 1991; Heckmann, 1993b] in connection with a general theory of relational databases. While these are quite natural from a domain-theoretic point of view, their equational characterizations (which do exist for some of them) are rather bizarre and do not give us much insight. The hyperspace approach is developed in logical form in Section 7.3.

We should also mention the various attempts to define a probabilistic version of the powerdomain construction, see [Saheb-Djahromi, 1980; Main, 1985; Graham, 1988; Jones and Plotkin, 1989; Jones, 1990]. (As an aside, these cannot be restricted to algebraic domains; the wider concept of continuous domain is forced upon us through the necessary use of the



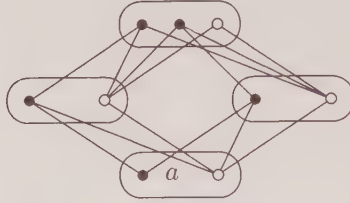
unit interval  $[0, 1]$ .) They do have an equational description in some sense but this goes beyond the techniques of this chapter.

One can then ask abstractly what constitutes a powerdomain construction and build a theory upon such a definition. This approach was taken by Heckmann [1990; 1991]. The most notable feature of this work is that under this perspective, too, many of the known powerdomains turn out to be canonical in a precise sense. How this (very natural) formulation of canonicity is connected with concerns in semantics, however, is as yet unclear.

### Exercises 6.2.23.

1. For the proof of Theorem 6.1.6 we can equip  $FB$  also with the transitive closure of  $\prec^s \circ \sqsubseteq$ . Show:
  - (a) This relation  $\prec'$  satisfies the interpolation axiom.
  - (b) In general,  $\prec'$  is different from  $\prec$ .
  - (c) The ideal completions of  $\langle FB, \prec \rangle$  and  $\langle FB, \prec' \rangle$  are isomorphic. (Use Exercise 2.3.9(27).)
  - (d) What is the advantage of  $\prec$  over  $\prec'$ ?
2. Describe the free domain algebra for an arbitrary domain  $D$  and an arbitrary signature  $\Sigma$  in the case that  $\mathcal{E}$  is empty.
3. Set up an algebraic theory such that all its dcpo-algebras have least elements but the embeddings  $\eta$  are not strict.
4. Let  $\langle \Sigma, \mathcal{E} \rangle$  be the usual equational theory of groups (or boolean algebras). Show that any dcpo-algebra with respect to this theory is trivially ordered. Conclude that the free construction collapses each connected component of the generating dcpo into a single point.
5. Given signatures  $\Sigma$  and  $\Sigma'$  and sets of inequalities  $\mathcal{E}$  and  $\mathcal{E}'$  we call the pair  $\langle \Sigma, \mathcal{E} \rangle$  a *reduct* of  $\langle \Sigma', \mathcal{E}' \rangle$  if  $\Sigma \subseteq \Sigma'$  and  $\mathcal{E} \subseteq \mathcal{E}'$ . In this case there is an obvious forgetful functor from  $\mathbf{C}(\Sigma', \mathcal{E}')$  to  $\mathbf{C}(\Sigma, \mathcal{E})$ , where  $\mathbf{C}$  is any of the categories considered in this chapter. Show that the general techniques of Theorem 6.1.2 and 6.1.7 suffice to prove that this functor has a left adjoint.
6. Likewise, show that partial domain algebras can be completed freely.
7. Let  $\underline{A}$  be a free domain-algebra over an algebraic domain. Is it true that every operation, if applied to compact elements of  $A$ , returns a compact element?
8. Let  $D = \{\perp \sqsubseteq a, b \sqsubseteq \top\}$  be the four-element lattice (Figure 1) and let  $E = D \times D$ . The sets  $\{\langle \perp, a \rangle, \langle \perp, b \rangle\}$  and  $\{\langle a, \perp \rangle, \langle b, \perp \rangle\}$  are elements of the Plotkin powerdomain of  $E$ . Show that they have two minimal upper bounds. Since  $\{\langle \top, \top \rangle\}$  is a top element,  $P^P(E)$  is not an L-domain.
9. Is the Plotkin powerdomain closed on  $\mathbf{F-B}$ , the category whose objects are bilimits of finite (but not necessarily pointed) posets?

10. Define a natural isomorphism between  $P^H(D)_{\perp} \multimap E$  and  $[D \rightarrow E]$  where  $D$  is any continuous domain,  $E$  is a complete lattice, and  $\multimap$  stands for the set of functions which preserve all suprema (ordered pointwise).



**Fig. 14.** Part of an algebraic domain where Theorem 6.2.19 fails.

11. We want to construct an algebraic domain  $D$  to which Theorem 6.2.19 cannot be extended. The compact elements of  $D$  are arranged in finite sets already such that they form a directed collection in the Egli–Milner ordering, generating the ideal  $I$ . We take one finite set for each element of  $\mathfrak{P}_f(\mathbb{R})$ , the finite powerset of the reals (or any other uncountable set), and we will have  $M_\alpha \ll_{EM} M_\beta$  if  $\alpha \subseteq \beta \subseteq \mathbb{R}$ . So we can arrange the  $M_\alpha$  in layers according to the cardinality of  $\alpha$ . Each  $M_\alpha$  contains one ‘white’ and  $|\alpha|!$  many ‘black’ elements. If  $\alpha \subsetneq \beta$  then the white element of  $M_\alpha$  is below every element of  $M_\beta$ . For the order between black elements look at adjacent layers. There are  $|\beta|$  many subsets of  $\beta$  with cardinality  $|\beta| - 1$ . The  $|\beta|!$  many black elements of  $M_\beta$  we partition into  $|\beta|$  many classes of cardinality  $(|\beta| - 1)!$ . So we can let the black elements of a lower neighbor of  $M_\beta$  be just below the equally many black elements of one of these classes. (The idea being that no two black elements have an upper bound.) Figure 14 shows a tiny fraction of the resulting ordered set  $K(D)$ . Establish the following facts about this domain:
- (a) Above a black element there are only black, below a white element there are only white elements.
  - (b)
    - i. An ideal in  $K(D)$  can contain at most one black element from each set.
    - ii. An ideal can contain at most one black element in each layer.
    - iii. An ideal can contain at most countably many black elements.
  - (c)
    - i. An ideal meeting all sets must contain all white elements.
    - ii. If an ideal contains a black element, then it contains the

least black element  $a$ .

- iii. If an ideal meeting all sets contains  $a$  then it must contain upper bounds for  $a$  and the uncountably many white elements of the first layer. These upper bounds must form an uncountable set and consist solely of black elements.
  - (d) From the contradiction between b-iii and c-iii conclude that only one ideal in  $K^D$  meets all sets, the ideal  $W$  of white elements. Therefore,  $\phi(I)$  contains precisely one element, say  $b$ . Show that  $\downarrow b$  equals  $W \cup \{b\}$  and that it is Scott-closed. Hence it is far from containing all elements of  $\bigcup I = K^D$ .
  - (e) Go a step farther and prove that the lenses of  $D$  are not even directed-complete by showing that the ideal  $I$  we started out with does not have an upper bound.
12. (R. Heckmann) Remove idempotence from the Hoare powertheory and study free domain algebras with respect to this theory. These are no longer finite if the generating domain is finite. Show that the free algebra over the four-element lattice (Figure 1) is neither bifinite nor an L-domain.

## 7 Domains and logic

There are at least three ways in which the idea of a function can be formalized. The first is via algorithms, which is the computer science viewpoint. The second is via value tables or, in more learned words, via graphs. This is the—rather recent—invention of mathematics. The third, finally, is via propositions: We can either take propositions about the function itself or view a function as something which maps arguments which satisfy  $\phi$  to values which satisfy  $\psi$ . The encoding in the latter case is by the set of all such pairs  $(\phi, \psi)$ . The beauty of the subject, then, lies in the interplay between these notions.

The passage from algorithms (programs) to the extensional description via graphs is called denotational semantics. It requires sophisticated structures, precisely *domains* in the sense of this text, because of, for example, recursive definitions in programs. The passage from algorithms to propositions about functions is called program logics. If we take the computer scientist's point of view as primary, then denotational semantics and program logics are two different ways of describing the behaviour of programs. It is the purpose of this chapter to lay out the connection between these two forms of semantics. As propositions we allow all those formulae whose extensions in the domain under consideration are (compact) Scott-open sets. This choice is well justified because it can be argued that such propositions correspond to properties which can be detected in a finite amount of time [Abramsky, 1987]. The reader will find lucid explications of this point in [Smyth, 1992] and [Vickers, 1989].

Mathematically, then, we have to study the relation between domains and their complete lattices of Scott-open sets. Stated for general topological spaces, this is the famous Stone duality. We treat it in Section 7.1. The restriction to domains introduces several extra features which we discuss one by one in Section 7.2. The actual domain logic, as a syntactical theory, is laid out in Section 7.3.

The whole open-set lattice, however, is too big to be syntactically represented. We must, on this higher level, once more employ ideas of approximation and bases. There is a wide range of possibilities here, which can be grouped under the heading of *information systems*. We concentrate on one of these, namely, the logic of compact open subsets. This is well-motivated by the general framework of Stone duality and also gives the richest logic.

## 7.1 Stone duality

### 7.1.1 Approximation and distributivity

We start out with a few observations concerning distributivity. So far, this has not played a role due to the poor order theoretic properties of domains. Now, in the context of open set lattices, it becomes a central theme, because, as we shall see, it is closely related with the concept of approximation. The earliest account of this connection is probably [Raney, 1953].

A word on notation: we shall try to keep a clear distinction between spaces, which in the end will be our domains, and their open-set lattices. We shall emphasize this by using  $\leq$  for the less-than-or-equal-to relation whenever we speak of lattices, even though these do form a special class of domains, too, as you may remember from Section 4.1.

Recall that a lattice  $L$  is said to be *distributive* if for all  $x, y, z \in L$  the equality

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

holds. The dual of this axiom is then satisfied as well. For the infinitary version of distributivity, we introduce the following notation for choice functions: If  $(A_i)_{i \in I}$  is a family of sets, then we write  $f: I \xrightarrow{\odot} \bigcup A_i$  if  $f(i)$  takes its value in  $A_i$  for every  $i \in I$ . *Complete distributivity* can then be expressed by the equation

$$\bigwedge_{i \in I} \bigvee A_i = \bigvee_{f: I \xrightarrow{\odot} \bigcup A_i} \bigwedge_{i \in I} f(i).$$

It, too, implies its order dual, see Exercise 7.3.19(1). There is a lot of room for variations of this and we shall meet a few of them in this section. Here comes the first:

**Theorem 7.1.1.** *A complete lattice  $L$  is continuous if and only if*



$$\bigwedge_{i \in I} \bigvee^\uparrow A_i = \bigvee^\uparrow_{f: I \xrightarrow{\odot} \bigcup A_i} \bigwedge_{i \in I} f(i)$$

holds for all families  $(A_i)_{i \in I}$  of directed subsets of  $L$ .

**Proof.** The reader should check for himself that the supremum on the right hand side is indeed over a directed set. Let now  $x$  be an element approximating the left hand side of the equation. Then for each  $i \in I$  we have  $x \ll \bigvee^\uparrow A_i$  and so there is  $a_i \in A_i$  with  $x \leq a_i$ . Let  $f$  be the choice function which selects these  $a_i$ . Then  $x \leq \bigwedge_{i \in I} f(i)$  and  $x$  is below the right hand side as well. Assuming  $L$  to be continuous, this proves  $\bigwedge_{i \in I} \bigvee^\uparrow A_i \leq \bigvee^\uparrow_{f: I \xrightarrow{\odot} \bigcup A_i} \bigwedge_{i \in I} f(i)$ . The reverse inequality holds in every complete lattice.

For the converse fix an element  $x \in L$  and let  $(A_i)_{i \in I}$  be the family of all directed sets  $A$  for which  $x \leq \bigvee^\uparrow A$ . From the equality, which we now assume to hold, we get that  $x = \bigvee^\uparrow_{f: I \xrightarrow{\odot} \bigcup A_i} \bigwedge_{i \in I} f(i)$ . We claim that for each choice function  $f: I \xrightarrow{\odot} \bigcup A_i$ , the corresponding element  $\bigwedge_{i \in I} f(i)$  is approximating  $x$ . Indeed, if  $A$  is a directed set with  $x \leq \bigvee^\uparrow A$  then  $A = A_{i_0}$  for some  $i_0 \in I$  and so  $\bigwedge_{i \in I} f(i) \leq f(i_0) \in A$ . ■

Let us now look at completely distributive lattices which, by the preceding theorem, are guaranteed to be continuous. We can go further and express this stronger distributivity by an approximation axiom, too.

**Definition 7.1.2.** For a complete lattice  $L$  define a relation  $\lll$  on  $L$  by

$$x \lll y \text{ if } \forall A \subseteq L. (y \leq \bigvee A \implies \exists a \in A. x \leq a).$$

Call  $L$  *prime-continuous* if for every  $x \in L$ ,  $x = \bigvee \{y \mid y \lll x\}$  holds.

Note that the relation  $\lll$  is defined in just the same way as the order of approximation, except that directed sets are replaced by arbitrary subsets. All our fundamental results about the order of approximation hold, *mutatis mutandis*, for  $\lll$  as well. In particular, we shall make use of Proposition 2.2.10 and Lemma 2.2.15. Adapting the previous theorem we get George N. Raney's characterization of complete distributivity [Raney, 1953].

**Theorem 7.1.3.** *A complete lattice is prime-continuous if and only if it is completely distributive.*

Let us now turn our attention to 'approximation' from above. The right concept for this is

**Definition 7.1.4.** A complete lattice  $L$  is said to be  $\wedge$ -generated by a subset  $A$  if for every  $x \in L$ ,  $x = \bigwedge (\uparrow x \cap A)$  holds. (Dually, we can speak of  $\vee$ -generation.)



We will study  $\wedge$ -generation by certain elements only, which we now introduce in somewhat greater generality than actually needed for our purposes.

**Definition 7.1.5.** An element  $x$  of a lattice  $L$  is called  $\wedge$ -irreducible if whenever  $x = \bigwedge M$  for a finite set  $M \subseteq L$  then it must be the case that  $x = m$  for some  $m \in M$ . We say  $x$  is  $\wedge$ -prime if  $x \geq \bigwedge M$  implies  $x \geq m$  for some  $m \in M$ , where  $M$  is again finite. Stating these conditions for arbitrary  $M \subseteq L$  gives rise to the notions of *completely  $\wedge$ -irreducible* and *completely  $\wedge$ -prime* element. The dual notions are obtained by exchanging supremum for infimum.

Note that neither  $\wedge$ -irreducible nor  $\wedge$ -prime elements are ever equal to the top element of the lattice, because that is the infimum of the empty set.

**Proposition 7.1.6.** *A  $\wedge$ -prime element is also  $\wedge$ -irreducible. The converse holds if the lattice is distributive.*

**Theorem 7.1.7.** *A continuous (algebraic) lattice  $L$  is  $\wedge$ -generated by its set of (completely)  $\wedge$ -irreducible elements.*

**Proof.** If  $x$  and  $y$  are elements of  $L$  such that  $x$  is not below  $y$  then there is a Scott-open filter  $F$  which contains  $x$  but not  $y$ , because  $\downarrow y$  is closed and the Scott-topology is generated by open filters, Lemma 2.3.8. Employing the Axiom of Choice in the form of Zorn's lemma, we find a maximal element above  $y$  in the inductive set  $L \setminus F$ . It is clearly  $\wedge$ -irreducible. In an algebraic lattice we can choose  $F$  to be a principal filter generated by a compact element. The maximal elements in the complement are then completely  $\wedge$ -irreducible. ■

**Theorem 7.1.8.** *If  $L$  is a complete lattice which is  $\wedge$ -generated by  $\wedge$ -prime elements, then  $L$  satisfies the equations*

$$\bigwedge_{m \in M} \bigvee A_m = \bigvee_{f: M \xrightarrow{\odot} \bigcup A_m} \bigwedge_{m \in M} f(m)$$

and

$$\bigvee_{i \in I} \bigwedge M_i = \bigwedge_{f: I \xrightarrow{\odot} \bigcup M_i} \bigvee_{i \in I} f(i)$$

where the sets  $M$  and  $M_i$  are finite.

A dual statement holds for lattices which are  $\vee$ -generated by  $\vee$ -prime elements.

**Proof.** The right-hand side is certainly below the left hand side, so assume that  $p$  is a  $\wedge$ -prime element above  $\bigvee_{f: M \xrightarrow{\odot} \bigcup A_m} \bigwedge_{m \in M} f(m)$ . Surely,  $p$  is above  $\bigwedge_{m \in M} f(m)$  for every  $f: M \xrightarrow{\odot} \bigcup A_m$  and because it is  $\wedge$ -prime it

is above  $f(m_f)$  for some  $M_f \in M$ . We claim that the set  $B$  of all  $f(m_f)$  covers at least one  $A_m$ . Assume the contrary. Then for each  $m \in M$  there exists  $a_m \in A_m \setminus B$  and we can define a choice function  $f_0: m \mapsto a_m$ . Then  $f_0(m_{f_0}) \in B$  contradicts our construction of  $f_0$ . So we know that for some  $m \in M$  all elements of  $A_m$  are below  $p$  and hence  $p$  is also above  $\bigwedge_{m \in M} \bigvee A_m$ . The proof for the second equation is similar and simpler. ■

Note that the two equations are not derivable from each other because of the side condition on finiteness. The first equation is equivalent to

$$x \wedge \bigvee_{i \in I} y_i = \bigvee_{i \in I} (x \wedge y_i)$$

which can be stated without choice functions. In this latter form it is known as the *frame distributivity law* and complete lattices which satisfy it are called *frames*. The basic operations on a frame are those which appear in this equation, namely, arbitrary join and finite meet.

### 7.1.2 From spaces to lattices

Given a topology  $\tau$  on a set  $X$  then  $\tau$  consists of certain subsets of  $X$ . We may think of  $\tau$  as an ordered set where the order relation is set inclusion. This ordered set is a complete lattice because arbitrary joins exist. Let us also look at continuous functions. In connection with open-set lattices it seems right to take the inverse image operation which, for a continuous function, is required to map opens to opens. Set-theoretically, it preserves all unions and intersections of subsets, and hence all joins and finite meets of opens. This motivates the following definition.

**Definition 7.1.9.** A *frame-homomorphism* between complete lattices  $K$  and  $L$  is a map which preserves arbitrary suprema and finite infima.

We let **CLat** stand for the category of complete lattices and frame-homomorphisms. We want to relate it to **Top**, the category of topological spaces and continuous functions. The first half of this relation is given by the contravariant functor  $\Omega$ , which assigns to a topological space its lattice of open subsets and to a continuous map the inverse image function.

For an alternative description let  $\mathbf{2}$  be the two-element chain  $\perp \leq \top$  equipped with the Scott-topology. The open sets of a space  $X$  are in one-to-one correspondence with continuous functions from  $X$  to  $\mathbf{2}$ , if for each open set  $O \subseteq X$  we set  $\chi_O$  to be the map which assigns  $\top$  to an element  $x$  if and only if  $x \in O$ . The action of  $\Omega$  on morphisms can then be expressed by  $\Omega(f)(\chi_O) = \chi_O \circ f$ .

### 7.1.3 From lattices to topological spaces

For motivation, let us look at topological spaces first. An element of a topological space  $X$  is naturally equipped with the following three pieces

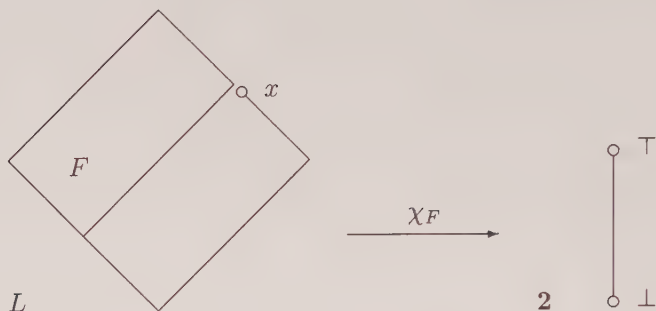


Fig. 15. A ‘point’ in a complete lattice.

of information. We can associate with it its filter  $\mathcal{F}_x$  of open neighborhoods, the complement of its closure, or a map from  $\mathbf{1}$ , the one-element topological space, to  $X$ . Taking the filter, for example, we observe that it has the additional property that if a union of open sets belongs to it then so does one of the opens. Also, the closure of a point has the property that it cannot be contained in a union of closed sets without being contained in one of them already. The map  $\mathbf{1} \rightarrow \mathbf{X}$ , which singles out the point, translates to a frame-homomorphism from  $\Omega(X)$  to  $\Omega(\mathbf{1}) = \mathbf{2}$ . Let us fix this new piece of notation:

**Definition 7.1.10.** A filter  $F \subseteq L$  is called *prime* if  $\bigvee M \in F$  implies  $F \cap M \neq \emptyset$  for all finite  $M \subseteq L$ . Allowing  $M$  to be an arbitrary subset we arrive at the notion of *completely prime filter*. Dually, we speak of (*completely*) *prime ideals*.

**Proposition 7.1.11.** Let  $L$  be a complete lattice and let  $F$  be a subset of  $L$ . The following are equivalent:

1.  $F$  is a completely prime filter.
2.  $F$  is a filter and  $L \setminus F = \downarrow x$  for some  $x \in L$ .
3.  $L \setminus F = \downarrow x$  for a  $\wedge$ -prime element  $x \in L$ .
4.  $\chi_F$  is a frame-homomorphism from  $L$  to  $\mathbf{2}$ .

This proposition shows that all three ways of characterizing points through opens coincide (see also Figure 15). Each of them has its own virtues and we will take advantage of the coincidence. As our official definition we choose the variant which is closest to our treatment of topological spaces.

**Definition 7.1.12.** Let  $L$  be a complete lattice. The *points* of  $L$  are the completely prime filters of  $L$ . The collection  $\text{pt}(L)$  of all points is turned

into a topological space by requiring all those subsets of  $\text{pt}(L)$  to be open which are of the form

$$\mathcal{O}_x = \{F \in \text{pt}(L) \mid x \in F\}, \quad x \in L.$$

**Proposition 7.1.13.** *The sets  $\mathcal{O}_x$ ,  $x \in L$ , form a topology on  $\text{pt}(L)$ .*

**Proof.** We have  $\bigcap_{m \in M} \mathcal{O}_{x_m} = \mathcal{O}_{\bigwedge_{m \in M} x_m}$ ,  $M$  finite, because points are filters and  $\bigcup_{i \in I} \mathcal{O}_{x_i} = \mathcal{O}_{\bigvee_{i \in I} x_i}$ , because they are completely prime. ■

Observe the perfect symmetry of our setup. In a topological space an element  $x$  belongs to an open set  $O$  if  $x \in O$ ; in a complete lattice a point  $F$  belongs to an open set  $\mathcal{O}_x$  if  $x \in F$ .

By assigning to a complete lattice  $L$  the topological space of all points, and to a frame-homomorphism  $h: K \rightarrow L$  the map  $\text{pt}(h)$  which assigns to a point  $F$  the point  $h^{-1}(F)$  (which is readily seen to be a completely prime filter), we get a contravariant functor, also denoted by  $\text{pt}$ , from  $\mathbf{CLat}$  to  $\mathbf{Top}$ .

Again, we give the alternative description based on characteristic functions. The fact is that we can use the same object **2** for this purpose, because it is a complete lattice as well. One speaks of a *schizophrenic object* in such a situation. As we saw in Proposition 7.1.11, a completely prime filter  $F$  gives rise to a frame-homomorphism  $\chi_F: L \rightarrow \mathbf{2}$ . The action of the functor  $\text{pt}$  on morphisms can then be expressed, as before, by  $\text{pt}(h)(\chi_F) = \chi_F \circ h$ .

#### 7.1.4 The basic adjunction

A topological space  $X$  can be mapped into the space of points of its open set lattice: simply map  $x \in X$  to the completely prime filter  $\mathcal{F}_x$  of its open neighborhoods. This assignment, which we denote by  $\eta_X: X \rightarrow \text{pt}(\Omega(X))$ , is continuous and open. Let  $U$  be an open set in  $X$ . Then we get by simply unwinding the definitions:  $\mathcal{F}_x \in \mathcal{O}_U \iff U \in \mathcal{F}_x \iff x \in U$ . It also commutes with continuous functions  $f: X \rightarrow Y$ :  $\text{pt}(\Omega(f))(\eta_X(x)) = \Omega(f)^{-1}(\mathcal{F}_x) = \mathcal{F}_{f(x)} = \eta_Y \circ f(x)$ . So the family of all  $\eta_X$  constitutes a natural transformation from the identity functor to  $\text{pt} \circ \Omega$ .

The same holds for complete lattices. We let  $\varepsilon_L: L \rightarrow \Omega(\text{pt}(L))$  be the map which assigns  $\mathcal{O}_x$  to  $x \in L$ . It is a frame-homomorphism as we have seen in the proof of Proposition 7.1.13. To see that this, too, is a natural transformation, we check that it commutes with frame-homomorphisms  $h: K \rightarrow L$ :  $\Omega(\text{pt}(h))(\varepsilon_K(x)) = \text{pt}(h)^{-1}(\mathcal{O}_x) = \mathcal{O}_{h(x)} = \varepsilon_L \circ h(x)$ , which is essentially the same calculation as for  $\eta$ . We have all the ingredients to formulate the Stone duality theorem:

**Theorem 7.1.14.** *The functors  $\Omega: \mathbf{Top} \rightarrow \mathbf{CLat}$  and  $\text{pt}: \mathbf{CLat} \rightarrow \mathbf{Top}$  are dual adjoints of each other. The units are  $\eta$  and  $\varepsilon$ .*

**Proof.** It remains to check the triangle equalities

$$\begin{array}{ccc}
 \Omega(X) & \xrightarrow{\varepsilon_{\Omega(X)}} & \Omega(\text{pt}(\Omega(X))) \\
 & \searrow \text{id} & \downarrow \Omega(\eta_X) \\
 & & \Omega(X)
 \end{array}
 \quad \text{and} \quad
 \begin{array}{ccc}
 \text{pt}(L) & \xrightarrow{\eta_{\text{pt}(L)}} & \text{pt}(\Omega(\text{pt}(L))) \\
 & \searrow \text{id} & \downarrow \text{pt}(\varepsilon_L) \\
 & & \text{pt}(L)
 \end{array}$$

For the left diagram let  $O$  be an open set in  $X$ .

$$\begin{aligned}
 \Omega(\eta_X)(\varepsilon_{\Omega(X)}(O)) &= \eta_X^{-1}(\mathcal{O}_O) &= \{x \in X \mid \eta_X(x) \in \mathcal{O}_O\} \\
 &= \{x \in X \mid \mathcal{F}_x \in \mathcal{O}_O\} \\
 &= \{x \in X \mid O \in \mathcal{F}_x\} \\
 &= \{x \in X \mid x \in O\} = O.
 \end{aligned}$$

The calculation for the right diagram is exactly the same if we exchange  $\eta$  and  $\varepsilon$ ,  $\Omega$  and  $\text{pt}$ ,  $X$  and  $L$ , and  $\mathcal{O}$  and  $\mathcal{F}$ . ■

While our concrete representation through open sets and completely prime filters, respectively, allowed us a very concise proof of this theorem, it is nevertheless instructive to see how the units behave in terms of characteristic functions. Their type is from  $X$  to  $(X \rightarrow \mathbf{2}) \rightarrow \mathbf{2}$  and from  $L$  to  $(L \rightarrow \mathbf{2}) \rightarrow \mathbf{2}$ , whereby the right-hand sides are revealed to be second duals. The canonical mapping into a second dual is, of course, point evaluation:  $x \mapsto \text{ev}_x$ , where  $\text{ev}_x(\chi) = \chi(x)$ . This is indeed what both  $\eta$  and  $\varepsilon$  do.

## 7.2 Some equivalences

### 7.2.1 Sober spaces and spatial lattices

In this subsection we look more closely at the units  $\eta$  and  $\varepsilon$ . We will need the following concept:

**Definition 7.2.1.** A closed subset of a topological space is called *irreducible* if it is non-empty and cannot be written as the union of two closed proper subsets.

Clearly, an irreducible closed set corresponds via complementation to a  $\wedge$ -irreducible (and hence  $\wedge$ -prime) element in the lattice of all open sets.

**Proposition 7.2.2.** *Let  $X$  be a topological space. Then  $\eta_X: X \rightarrow \text{pt}(\Omega(X))$  is injective if and only if  $X$  satisfies the  $T_0$ -separation axiom. It is surjective if and only if every irreducible closed set is the closure of an element of  $X$ .*

**Proof.** The first half is just one of the various equivalent definitions of  $T_0$ -separation: different elements have different sets of open neighbourhoods.



For the second statement observe that the  $\wedge$ -prime elements of  $\Omega(X)$  are in one-to-one correspondence with completely prime filters of open sets. The condition then simply says that every such filter arises as the neighbourhood filter of an element of  $X$ . ■

**Definition 7.2.3.** A topological space  $X$  is called *sober* if  $\eta_X$  is bijective.

Note that if  $\eta_X$  is bijective then it must be a homeomorphism because we know from Section 7.1.4 that it is always continuous and open. By the preceding proposition, a space is sober if and only if it is  $T_0$  and every irreducible closed set is the closure of a point. The intuitive meaning is, of course, that a space is sober if it can be recovered from its lattice of open sets.

**Proposition 7.2.4.** For any complete lattice  $L$  the unit  $\varepsilon_L: L \rightarrow \Omega(\text{pt}(L))$  is surjective and monotone. Furthermore, the following are equivalent:

1.  $\varepsilon_L$  is injective.
2. The elements of  $L$  are separated by completely prime filters.
3.  $L$  is  $\wedge$ -generated by  $\wedge$ -prime elements.
4. If  $x \not\leq y$  then there exists a completely prime filter  $F$  such that  $x \in F$  and  $y \notin F$ .
5.  $\varepsilon_L$  is order-reflecting.

**Proof.** We have seen in Proposition 7.1.13 that all open sets on  $\text{pt}(L)$  are of the form  $\mathcal{O}_x$  for some  $x \in L$ . This proves surjectivity. Monotonicity is clear because filters are upper sets.

Turning to the equivalent conditions for injectivity, we note that  $\mathcal{O}_x = \mathcal{O}_y$  is equivalent to  $x \in F \iff y \in F$  for all completely prime filters  $F$ . In other words,  $\varepsilon_L$  is injective if and only if the elements of  $L$  are separated by completely prime filters. Given  $x \in L$  let  $x'$  be the infimum of all  $\wedge$ -primes above  $x$ . We want to show that  $x = x'$ . If  $x'$  is strictly above  $x$  then there exists a completely prime filter containing  $x'$  but not  $x$ . Using the equivalence of Proposition 7.1.11, we see that this is the same as the existence of a  $\wedge$ -prime element in  $\uparrow x \setminus \uparrow x'$ , a contradiction. From (3) the last two statements follow easily. They, in turn, imply injectivity (which, in a general order-theoretic setting, is strictly weaker than order-reflection). ■

**Definition 7.2.5.** A complete lattice  $L$  is called *spatial* if  $\varepsilon_L$  is bijective.

The intuitive meaning in this case is that a spatial lattice can be thought of as a lattice of open sets for some topological space. A direct consequence of Theorem 7.1.8 is the following:

**Theorem 7.2.6.** A spatial lattice is a frame. In particular, it is distributive.

**Theorem 7.2.7.** For any complete lattice  $L$  the topological space  $\text{pt}(L)$  is sober. For any topological space  $X$  the lattice  $\Omega(X)$  is spatial.

**Proof.** The space of points of a lattice  $L$  is certainly  $T_0$ , because if we are given different completely prime filters then there is  $x \in L$  which belongs to one of them but not the other. Hence,  $\mathcal{O}_x$  contains one but not the other. For surjectivity of  $\eta_{\text{pt}(L)}$  let  $\mathcal{A}$  be an irreducible closed set of filters. First of all, the union  $A$  of all filters in  $\mathcal{A}$  is a non-empty upper set in  $L$  which is unreachable by joins. Hence the complement of  $A$  is a principal ideal  $\downarrow x$ . Also, the complement of  $\mathcal{A}$  in  $\text{pt}(L)$  certainly contains  $\mathcal{O}_x$ . We claim that  $x$  must be  $\wedge$ -prime. Indeed, if  $y \wedge z \leq x$  then  $\mathcal{A}$  is covered by the complements of  $\mathcal{O}_y$  and  $\mathcal{O}_z$ , whence it is covered by one of them, say the complement of  $\mathcal{O}_y$ , which means nothing else than  $y \leq x$ . It follows that  $\mathcal{A}$  is contained in the closure of the point  $L \setminus \downarrow x$ . On the other hand,  $L \setminus \downarrow x$  belongs to the closed set  $\mathcal{A}$  as each of its open neighbourhoods contains an element of  $\mathcal{A}$ .

The second statement is rather easier to argue for. If  $O$  and  $O'$  are different open sets then there is an element  $x$  of  $X$  contained in one but not the other. Hence the neighbourhood filter of  $x$ , which is always completely prime, separates  $O$  and  $O'$ . ■

**Corollary 7.2.8.** *The functors  $\Omega$  and  $\text{pt}$  form a dual equivalence between the category of sober spaces and the category of spatial lattices.*

This result may suggest that a reasonable universe of topological spaces ought to consist of sober spaces, or, if one prefers the lattice-theoretic side, of spatial lattices. This is indeed true as far as spaces are concerned. For the lattice side, however, it has been argued forcefully that the right choice is the larger category of *frames* (which are defined to be those complete lattices which satisfy the frame distributivity law, Section 7.1.1). The basis of these arguments is the fact that free frames exist, see [Johnstone, 1982], Theorem II.1.2, a property which holds neither for complete lattices nor for spatial lattices. (More information on this is in [Isbell, 1972; Johnstone, 1982; Johnstone, 1983].) The choice of using frames for doing topology has more recently found support from theoretical computer science, because it is precisely the frame distributivity law which can be expected to hold for observable properties of processes. Even though this connection is to a large extent the *raison d'être* for this chapter, we must refer to [Abramsky, 1987; Abramsky, 1991b; Vickers, 1989; Smyth, 1992] for an in-depth discussion.

## 7.2.2 Properties of sober spaces

Because application of  $\text{pt} \circ \Omega$  to a space  $X$  is an essentially idempotent operation, it is best to think of  $\text{pt}(\Omega(X))$  as a completion of  $X$ . It is commonly called the *soberification* of  $X$ . Completeness of this particular kind is also at the heart of the Hofmann–Mislove theorem, which we have met in Section 4.2.3 already and which we are now able to state in its full generality.

**Theorem 7.2.9.** *Let  $X$  be a sober space. The sets of open neighbourhoods*

of compact saturated sets are precisely the Scott-open filters in  $\Omega(X)$ .

**Proof.** It is pretty obvious that the neighbourhoods of compact subsets are Scott-open filters in  $\Omega(X)$ . We are interested in the other direction. Given a Scott-open filter  $\mathcal{F} \subseteq \Omega(X)$  then the candidate for the corresponding compact set is  $K = \bigcap \mathcal{F}$ . We must show that each open neighbourhood of  $K$  belongs to  $\mathcal{F}$  already. For the sake of contradiction assume that there exists an open neighborhood  $O \notin \mathcal{F}$ . By Zorn's lemma we may further assume that  $O$  is maximal with this property. Because  $\mathcal{F}$  is a filter,  $O$  is  $\wedge$ -prime as an element of  $\Omega(X)$  and this is tantamount to saying that its complement  $A$  is irreducible as a closed set. By sobriety it must be the closure of a single point  $x \in X$ . The open sets which do not contain  $x$  are precisely those which are contained in  $O$ . Hence every open set from the filter  $\mathcal{F}$  contains  $x$  and so  $x$  belongs to  $K$ . This, finally, contradicts our assumption that  $O$  is a neighborhood of  $K$ . ■

This appeared first in [Hofmann and Mislove, 1981]. Our proof is taken from [Keimel and Paseka, to appear]. Note that it relies, like almost everything else in this chapter, on the axiom of choice.

Saturated sets are uniquely determined by their open neighbourhoods, so we can reformulate the preceding theorem as follows:

**Corollary 7.2.10.** *Let  $X$  be a sober space. The poset of compact saturated sets ordered by inclusion is dually isomorphic to the poset of Scott-open filters in  $\Omega(X)$  (also ordered by inclusion).*

**Corollary 7.2.11.** *Let  $X$  be a sober space. The filtered intersection of a family of (non-empty) compact saturated subsets is compact (and non-empty). If such a filtered intersection is contained in an open set  $O$  then some element of the family belongs to  $O$  already.*

**Proof.** By the Hofmann–Mislove theorem we can switch freely between compact saturated sets and open filters in  $\Omega(X)$ . Clearly, the directed union of open filters is another such. This proves the first statement. For the intersection of a filtered family to be contained in  $O$ ,  $O$  must belong to the directed union of the corresponding filters. Then  $O$  must be contained in one of these. The claim about the intersection of non-empty sets follows from this directly because we can take  $O = \emptyset$ . ■

Every  $T_0$ -space can be equipped with an order relation, called the *specialization order*, by setting  $x \sqsubseteq y$  if for all open sets  $O$ ,  $x \in O$  implies  $y \in O$ . We may then compare the given topology with topologies defined on ordered sets. One of these which plays a role in this context, is the *weak upper topology*. It is defined as the coarsest topology for which all sets of the form  $\downarrow x$  are closed.

**Proposition 7.2.12.** *For a  $T_0$ -space  $X$  the topology on  $X$  is finer than the weak upper topology derived from the specialization order.*

**Proposition 7.2.13.** *A sober space is a dcpo in its specialization order and its topology is coarser than the Scott-topology derived from this order.*

**Proof.** By the equivalence between sober spaces and spatial lattices we may think of  $X$  as the points of a complete lattice  $L$ . It is seen without difficulty that the specialization order on  $X$  then translates to the inclusion order of completely prime filters. That a directed union of completely prime filters is again a completely prime filter is immediate.

Let  $\bigcup_{i \in I}^\uparrow F_i$  be such a directed union. It belongs to an open set  $\mathcal{O}_x$  if and only if  $x \in F_i$  for some  $i \in I$ . This shows that each  $\mathcal{O}_x$  is Scott-open. ■

A dcpo equipped with the Scott-topology, on the other hand, is not necessarily sober, see Exercise 7.3.19(7). We also record the following fact, although we shall not make use of it.

**Theorem 7.2.14.** *The category of sober spaces is complete and cocomplete. It is also closed under retracts formed in the ambient category **Top**.*

For the reader's convenience we sum up our considerations in Table 2, comparing concepts in topological spaces to concepts in  $\text{pt}(L)$  for  $L$  a complete lattice.

space	$\text{pt}(L)$
point	completely prime filter (c. p. filter)
specialization order	inclusion order
open set	c. p. filters containing some $x \in L$
saturated set	c. p. filters containing some upper set
compact saturated set	c. p. filters containing a Scott-open filter

**Table 2.** Topological spaces compared with  $\text{pt}(L)$ .

### 7.2.3 Locally compact spaces and continuous lattices

We already know that sober spaces may be seen as dcpos with an order-consistent topology. We move on to more special kinds of spaces with the aim of characterizing our various kinds of domains through their open-set lattices. Our first step in this direction is to introduce local compactness. We have

**Lemma 7.2.15.** *Distributive continuous lattices are spatial.*

**Proof.** We have shown in Theorem 7.1.7 that continuous lattices are  $\wedge$ -generated by  $\wedge$ -irreducible elements. In a distributive lattice these are also  $\wedge$ -prime. ■

Now recall that a topological space is called locally compact if every element has a fundamental system of compact neighbourhoods. This alone does not imply sobriety, as the ascending chain of natural numbers,



equipped with the weak upper topology, shows. But in combination with sobriety we get the following beautiful result:

**Theorem 7.2.16.** *The functors  $\Omega$  and  $\text{pt}$  restrict to a dual equivalence between the category of sober locally compact spaces and the category of distributive continuous lattices.*

**Proof.** We have seen in Section 4.2.3 already that  $O \ll O'$  holds in  $\Omega(X)$  if there is a compact set between  $O$  and  $O'$ . This proves that the open-set lattice of a locally compact space is continuous.

For the converse, let  $F$  be a point in an open set  $\mathcal{O}_x$ , that is,  $x \in F$ . A completely prime filter is Scott-open, therefore there is a further element  $y \in F$  with  $y \ll x$ . Lemma 2.3.8 tells us that there is a Scott-open filter  $G$  contained in  $\uparrow y$  which contains  $x$ . We know by the previous lemma that a distributive continuous lattice can be thought of as the open-set lattice of its space of points, which, furthermore, is guaranteed to be sober. So we can apply the Hofmann–Mislove Theorem 7.2.9 and get that the set  $\mathcal{A}$  of points of  $L$ , which are supersets of  $G$ , is compact saturated. In summary,  $F$  is contained in  $\mathcal{O}_y$ , which is a subset of  $\mathcal{A}$ , and this is a subset of  $\mathcal{O}_x$ . ■

From now on, all our spaces are locally compact and sober. The three properties introduced in the next three subsections, however, are independent of each other.

## 7.2.4 Coherence

We have introduced coherence in Section 4.2.3 for the special case of continuous domains. The general definition reads as follows:

**Definition 7.2.17.** A topological space is called *coherent*, if it is sober, locally compact, and the intersection of two compact saturated subsets is compact.

**Definition 7.2.18.** The order of approximation on a complete lattice is called *multiplicative* if  $x \ll y$  and  $x \ll z$  imply  $x \ll y \wedge z$ . A distributive continuous lattice for which the order of approximation is multiplicative is called *arithmetic*.

As a generalization of Proposition 4.2.16 we have

**Theorem 7.2.19.** *The functors  $\Omega$  and  $\text{pt}$  restrict to a dual equivalence between the category of coherent spaces and the category of arithmetic lattices.*

**Proof.** The same arguments as in Proposition 4.2.15 apply, so it is clear that the open-set lattice of a coherent space is arithmetic. For the converse we may, just as in the proof of Theorem 7.2.19, invoke the Hofmann–Mislove theorem. It tells us that compact saturated sets of  $\text{pt}(L)$  are in one-to-one correspondence with Scott-open filters. Multiplicativity of the



order of approximation is just what we need to prove that the pointwise infimum of two Scott-open filters is again Scott-open. ■

### 7.2.5 Compact-open sets and spectral spaces

By passing from continuous lattices to algebraic ones we get:

**Theorem 7.2.20.** *The functors  $\Omega$  and  $\text{pt}$  restrict to a dual equivalence between the category of sober spaces, in which every element has a fundamental system of compact-open neighbourhoods, and the category of distributive algebraic lattices.*

The proof is the same as for distributive continuous lattices, Theorem 7.2.16. We now combine this with coherence.

**Definition 7.2.21.** A topological space, which is coherent and in which every element has a fundamental system of compact-open neighbourhoods, is called a *spectral space*.

**Theorem 7.2.22.** *The functors  $\Omega$  and  $\text{pt}$  restrict to a dual equivalence between the category of spectral spaces and the category of algebraic arithmetic lattices.*

Having arrived at this level, we can replace the open-set lattice with the sublattice of compact-open subsets. Our next task then is to reformulate Stone-duality with bases of open-set lattices. For objects we have

**Proposition 7.2.23.** *Let  $L$  be an algebraic arithmetic lattice. The completely prime filters of  $L$  are in one-to-one correspondence with the prime filters of  $K(L)$ . The topology on  $\text{pt}(L)$  is generated by the set of all  $\mathcal{O}_x$ , where  $x$  is compact in  $L$ .*

**Proof.** Given a completely prime filter  $F$  in  $L$ , we let  $F \cap K(L)$  be the set of compact elements contained in it. This is clearly an upwards closed set in  $K(L)$ . It is a filter, because  $L$  is arithmetic. Primeness, finally, follows from the fact that  $F$  is Scott-open and hence equal to  $\uparrow(F \cap K(L))$ . Conversely, a filter  $G$  in  $K(L)$  generates a filter  $\uparrow G$  in  $L$ . For complete primeness let  $A$  be a subset of  $L$  with join in  $\uparrow G$ .  $L$  is algebraic. So we may replace  $A$  by  $B = \downarrow A \cap K(L)$  and  $\bigvee B \in \uparrow G$  will still hold. Because  $\uparrow G$  is Scott-open, there is a finite subset  $M$  of  $B$  with  $\bigvee M \in \uparrow G$ . Some element of  $G$  must be below  $\bigvee M$  and primeness then gives us that some element of  $M$  belongs to  $G$ .

The statement about the topology on  $\text{pt}(L)$  follows from the fact that every element of  $L$  is a join of compact elements. ■

A frame-homomorphism between algebraic arithmetic lattices need not preserve compact elements, so in order to represent it through bases we need to resort to relations, as in Section 2.2.6, Definition 2.2.27. Two additional axioms are needed, however, because frame-homomorphisms are more special than Scott-continuous functions.

**Definition 7.2.24.** A relation  $R$  between lattices  $V$  and  $W$  is called *join-approximable* if the following conditions are satisfied:

1.  $\forall x, x' \in V \forall y, y' \in W. (x' \geq x \ R \ y \geq y' \implies x' \ R \ y');$
2.  $\forall x \in V \forall N \subseteq_{\text{fin}} W. (\forall y \in N. x \ R \ y \implies x \ R \ (\bigvee N));$
3.  $\forall M \subseteq_{\text{fin}} V \forall y \in W. (\forall x \in M. x \ R \ y \implies (\bigwedge M) \ R \ y);$
4.  $\forall M \subseteq_{\text{fin}} V \forall y \in W. ((\bigvee M) \ R \ y \implies \exists N \subseteq_{\text{fin}} W. (y = \bigvee N \wedge \forall n \in N \exists m \in M. m \ R \ n)).$

The following is then easily established:

**Proposition 7.2.25.** *The category of algebraic arithmetic lattices and frame-homomorphisms is equivalent to the category of distributive lattices and join-approximable relations.*

By Proposition 7.2.23 we can replace the compound functor  $\text{pt} \circ \text{Idl}$  by a direct construction of a topological space out of a distributive lattice. We denote this functor by  $\text{spec}$ , standing for the *spectrum* of a distributive lattice. We also contract  $K \circ \Omega$  to  $K\Omega$ . Then we can say

**Theorem 7.2.26.** *The category of spectral spaces and continuous functions is dually equivalent to the category of distributive lattices and join-approximable relations via the contravariant functors  $K\Omega$  and  $\text{spec}$ .*

We supplement the Table 2 in Section 7.2.2 with the following comparison of concepts in a topological space and concepts in the spectrum of a distributive lattice.

space	$\text{spec}(L)$
point	prime filter
specialization order	inclusion order
compact-open set	prime filters containing some $x \in L$
open set	union of compact open sets
saturated set	prime filters containing some upper set
compact saturated set	prime filters containing a filter

**Table 3.** Topological spaces compared with  $\text{spec}(L)$ .

It has been argued that the category of spectral spaces is the right setting for denotational semantics, precisely because these have a finitary ‘logical’ description through their distributive lattices of compact-open subsets, see [Smyth, 1992], for example. However, this category is neither cartesian closed, nor does it have fixpoints for endofunctions, and hence does not provide an adequate universe for the semantics of computation. An intriguing question arises, of how the kinds of spaces traditionally studied in topology and analysis can best be reconciled with the computational intuitions reflected in the very different kinds of spaces which arise in domain theory. An interesting recent development is Abbas Edalat’s use of

domain theory as the basis for a novel approach to the theory of integration [Edalat, 1993a].

### 7.2.6 Domains

Let us now see how continuous domains come into the picture. First we note that sobriety no longer needs to be assumed:

**Proposition 7.2.27.** *Continuous domains equipped with the Scott-topology are sober spaces.*

**Proof.** Let  $A$  be an irreducible closed set in a continuous domain  $D$  and let  $B = \downarrow A$ . We show that  $B$  is directed. Indeed, given  $x$  and  $y$  in  $B$ , then neither  $D \setminus \uparrow x$  nor  $D \setminus \uparrow y$  contain all of  $A$ . By irreducibility, then, they can't cover  $A$ . Hence there is  $a \in A \cap \uparrow x \cap \uparrow y$ . But since  $\uparrow x \cap \uparrow y$  is Scott-open, there is also some  $b \ll a$  in this set. This gives us the desired upper bound for  $x$  and  $y$ . It is plain from Proposition 2.2.10 that  $A$  is the closure of  $\bigcup \uparrow B$ . ■

The following result of Jimmie Lawson and Rudolf-Eberhard Hoffmann, [Lawson, 1979; Hoffmann, 1981], demonstrates once again the central role played by continuous domains.

**Theorem 7.2.28.** *The functors  $\Omega$  and  $\text{pt}$  restrict to a dual equivalence between **CONT** and the category of completely distributive lattices.*

**Proof.** A Scott-open set  $O$  in a continuous domain  $D$  is a union of sets of the form  $\uparrow x$  where  $x \in O$ . For each of these we have  $\uparrow x \lll O$  in  $\sigma_D$ . This proves complete distributivity, as we have seen in Theorem 7.1.3.

For the converse, let  $L$  be completely distributive. We already know that the points of  $L$  form a dcpo (where the order is given by inclusion of filters) and that the topology on  $\text{pt}(L)$  is contained in the Scott-topology of this dcpo. Now we show that every completely prime filter  $F$  has enough approximants. Observe that  $F' \ll F$  certainly holds in all those cases where  $\bigwedge F'$  is an element of  $F$  as directed suprema of points are unions of filters. Now given  $x \in F$  we get from prime-continuity that  $x = \bigvee \{y \mid y \lll x\}$  and so there must be some  $y \in F$  with  $y \lll x$ . Successively interpolating between  $y$  and  $x$  gives us a sequence of elements such that  $y \lll \dots \lll y_n \lll \dots \lll y_1 \lll x$ , just as in the proof of Lemma 2.3.8. The set  $\bigcup_{n \in \mathbb{N}} \uparrow y_n$  then is a completely prime filter containing  $x$  with infimum in  $F$ . The directedness of these approximants is clear because  $F$  is filtered. As a consequence, we have that  $F' \ll F$  holds if and only if  $\bigwedge F'$  belongs to  $F$ .

We are not quite finished, though, because we also need to show that we get the Scott-topology back. To this end let  $\mathcal{O}$  be a Scott-open set of points, that is,  $F \supseteq F' \in \mathcal{O}$  implies  $F \in \mathcal{O}$  and  $\bigcup_{i \in I} F_i \in \mathcal{O}$  implies  $F_i \in \mathcal{O}$  for some  $i \in I$ . Let  $x$  be the supremum of all elements of the form  $\bigwedge F$ ,  $F \in \mathcal{O}$ . We claim that  $\mathcal{O} = \mathcal{O}_x$ . First of all, for each  $F \in \mathcal{O}$  there

is  $F' \in \mathcal{O}$  with  $F' \ll F$ , which, as we have just seen, is tantamount to  $\bigwedge F' \in F$ , hence  $x$  belongs to all  $F$  and  $\mathcal{O} \subseteq \mathcal{O}_x$  is proved.

Conversely, if a point  $G$  contains  $x$  then it must contain some  $\bigwedge F$ ,  $F \in \mathcal{O}$ , because it is completely prime. Hence  $G$  belongs to  $\mathcal{O}$ , too, and we have shown  $\mathcal{O}_x \subseteq \mathcal{O}$ . ■

To this we can add coherence and we get a dual equivalence between coherent domains and completely distributive arithmetic lattices. Or we can add algebraicity and get a dual equivalence between algebraic domains and algebraic completely distributive lattices. Adding both properties characterizes what can be called 2/3-bifinite domains in the light of Proposition 4.2.17. We prefer to speak of coherent algebraic domains. As these are spectral spaces, we may also ask how they can be characterized through the lattice of compact open subsets. The answer is rather simple: a compact open set in an algebraic domain  $D$  is a finite union of sets of the form  $\uparrow c$  for  $c \in K(D)$ . These, in turn, are characterized by being  $\vee$ -irreducible and also  $\vee$ -prime.

**Theorem 7.2.29.** *The dual equivalence of Theorem 7.2.26 cuts down to a dual equivalence of coherent algebraic domains and lattices in which every element is the join of finitely many  $\vee$ -primes.*

**Proof.** We only need to show that if a lattice satisfies the condition stated in the theorem, then its ideal completion is completely distributive. But this is trivial because a principal ideal generated by a  $\vee$ -prime is completely  $\vee$ -prime in the ideal completion and so the result follows from Theorem 7.1.3. ■

All the combined strength of complete distributivity, algebraicity and multiplicativity of the order of approximation, however, still does not restrict the corresponding spaces far enough to bring us into one of our cartesian closed categories of domains. Let us therefore see what we have to add in order to characterize bifinite domains. The only solution in this setting appears to be a translation of mub-closures into the lattice of compact-open subsets, that is to say, the subset of  $\vee$ -primes has the upside-down finite mub property (Definition 4.2.1). Let us sum up these considerations in a theorem:

**Theorem 7.2.30.** *A lattice  $V$  is isomorphic to the lattice of compact-open subsets of an  $F$ - $B$ -domain (Definition 4.3.7) if and only if, firstly,  $V$  has a least element, secondly, each element of  $V$  is the supremum of finitely many  $\vee$ -primes and, thirdly, for every finite set  $M$  of  $\vee$ -primes there is a finite superset  $N$  of  $\vee$ -primes such that*

$$\forall A \subseteq N \exists B \subseteq N. \bigwedge A = \bigvee B.$$



*The additional requirement that there be a largest element which is also  $\vee$ -prime, characterizes the lattices of compact-open subsets of bifinite domains.*

The extra condition about finite mub-closures is not a first-order axiom and cannot be replaced by one as was shown by Carl Gunter in [Gunter, 1986]. The smaller class of algebraic bc-domains has a rather nicer description:

**Theorem 7.2.31.** *A lattice  $V$  is isomorphic to the lattice of compact-open subsets of an algebraic bc-domain if and only if it has a least element, each element of  $V$  is the supremum of finitely many  $\vee$ -primes and the set of  $\vee$ -primes plus least element is closed under finite infima.*

### 7.2.7 Summary

We have summarized the results of this section in Figure 16 and Table 4. As labels we have invented a few mnemonic names for categories. We won't use them outside this subsection. The filled dots correspond to categories for which there is also a characterization in terms of compact-open subsets (spectral spaces). A similar diagram appears in [Gierz *et al.*, 1980] but there not everything which appears to be an intersection of categories really is one.

## 7.3 The logical viewpoint

This material is based on [Abramsky, 1991b].

### 7.3.1 Working with lattices of compact-open subsets

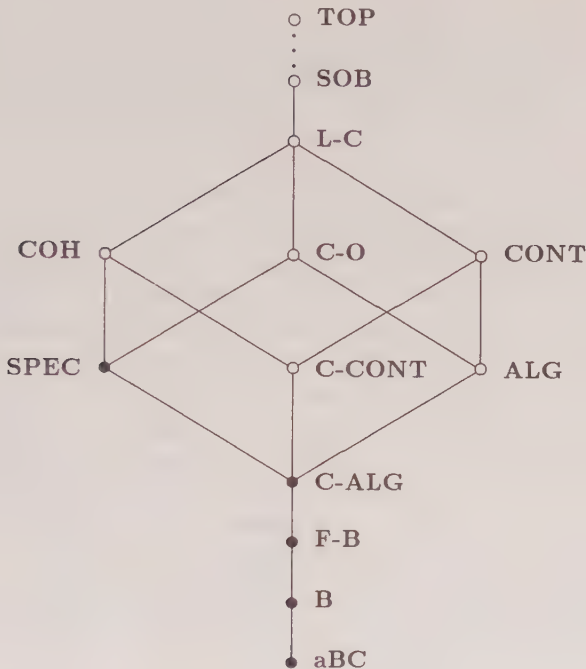
Having established the duality between algebraic domains and their lattices of compact-open subsets we can now ask to what extent we can do domain theory through these lattices. We have already indicated that such an approach offers many new insights but for the moment our motivation could simply be that working with lattices is a lot easier than working with dcpo's. 'Doing domain theory' refers to performing the domain constructions of Sections 3.2, 3.3, 5 and 6, at least in a first approximation.

Let us try this out. Suppose you know  $K\Omega(D)$  for some bifinite domain  $D$ , how do you construct  $K\Omega(D_\perp)$ , the lattice of compact-open subsets of the lifted domain? The answer is simple: just add a new top element,  $K\Omega(D_\perp) = K\Omega(D)^\top$ . Coalesced sum also works fine:

$$K\Omega(D \oplus E) = (K\Omega(D) \setminus \{D\}) \times (K\Omega(E) \setminus \{E\}) \cup \{D \oplus E\}.$$

We encounter the first problems when we look at the cartesian product. While it is clear that every compact-open subset of  $D \times E$  is a finite union of products of compact-open subsets in the factors, there seems to be no simple criterion on such unions which would guarantee unique representation.





- TOP** Topological spaces. No Stone-dual.
- SOB** Sober spaces vs. spatial lattices.
- L-C** Locally-compact sober spaces vs. continuous distributive lattices.
- COH** Coherent spaces (= locally compact, sober, and intersection of compact saturated is compact) vs. arithmetic lattices (= distributive, continuous, and order of approximation is multiplicative).
- C-O** Sober spaces with a base of compact-open sets vs. distributive algebraic lattices.
- CONT** Continuous domains with Scott-topology vs. completely distributive lattices.
- SPEC** Spectral spaces vs. algebraic arithmetic lattices vs. distributive lattices.
- C-CONT** Coherent domains vs. arithmetic completely distributive lattices.
- ALG** Algebraic domains vs. algebraic completely distributive lattices.
- C-ALG** Coherent algebraic domains vs. algebraic arithmetic completely distributive lattices vs. distributive lattices in which every element is the finite join of  $\vee$ -primes.
- F-B** F-B-domains (Definition 4.3.7) (= bilimits of finite posets). Stone-dual only described through the basis (or base) of compact-open subsets, which is a distributive lattice with extra properties as stated in Theorem 7.2.30.
- B** Bifinite domains. Stone-dual only described through the basis of compact-open subsets, which is a distributive lattice with extra properties as stated in Theorem 7.2.30.
- aBC** Algebraic bounded-complete domains. Stone-dual only described through the basis of compact-open subsets, which is a distributive lattice with extra properties as stated in Theorem 7.2.31.

Fig. 16. An overview of Stone-dualities in domain theory.

The moral then is that we must allow for multiple representations of compact-open subsets. Instead of lattices we shall study certain preordered structures. At first glance this may seem an unwanted complication but we will soon see that it really makes the whole programme work much more smoothly.

Lattices are determined by either their order structure or their algebraic structure but this equivalence no longer holds in the preordered case. Instead we must mention both preorder and lattice operations. We also make  $\vee$ -primeness explicit in our axiomatization. The reason for this is that we want to keep all our definitions inductive. This point will become clearer when we discuss the function space construction below.

**Definition 7.3.1.** A *coherent algebraic prelocale*  $A$  is a preordered algebra with two binary operations  $\vee$  and  $\wedge$ , two nullary operations  $0$  and  $1$ , and a unary predicate  $C$  on  $A$ , such that  $a \vee b$  is a supremum for  $\{a, b\}$ ,  $a \wedge b$  is an infimum for  $\{a, b\}$ ,  $0$  is a least, and  $1$  is a largest element. The preorder on  $A$  is denoted by  $\lesssim$ , the corresponding equivalence relation by  $\approx$ . The predicate  $C(a)$  is required to hold if and only if  $a$  is  $\vee$ -prime. Finally, every element of  $A$  must be equivalent to a finite join of  $\vee$ -primes.

We will not distinguish between a prelocale and its underlying set. The set  $\{a \in A \mid C(a)\}$  is abbreviated as  $C(A)$ .

This is essentially the definition which appears in [Abramsky, 1991b]. There another predicate is included. We can omit this because we will not look at the coalesced sum construction. The expressions ‘a supremum’, ‘an infimum’, etc., may seem contradictory but they are exactly appropriate in the preordered universe. It is seen without difficulty that every coherent algebraic prelocale  $A$  gives rise to a lattice  $A/\approx$  which is  $\vee$ -generated by  $\vee$ -primes and hence distributive.

A *domain prelocale* is obtained by incorporating the two extra conditions from Theorem 7.2.30:

- $\forall u \subseteq_{\text{fin}} C(A) \exists v \subseteq_{\text{fin}} C(A). u \subseteq v \text{ and } (\forall w \subseteq v \exists z \subseteq v. \bigwedge w = \bigvee z);$
- $C(1).$

**Definition 7.3.2.** Let  $A$  and  $B$  be domain prelocales. A function  $\phi: A \rightarrow B$  is called a *pre-isomorphism* if it is surjective, order-preserving and order-reflecting. If  $A$  is a domain prelocale and  $D$  is a bifinite domain and if further there is a pre-isomorphism  $\llbracket \cdot \rrbracket: A \rightarrow K\Omega(D)$ , then we say that  $A$  is a *localic description* of  $D$  via  $\llbracket \cdot \rrbracket$ .

A pre-isomorphism  $\phi: A \rightarrow B$  must preserve suprema, infima, and least and largest element (up to equivalence). Furthermore, it restricts and corestricts to a surjective map  $\phi^0: C(A) \rightarrow C(B)$ . Let us look more closely at the case of a pre-isomorphism  $\llbracket \cdot \rrbracket: A \rightarrow K\Omega(D)$ . A diagram may be quite helpful:

$$\begin{array}{ccc}
C(A) & \hookrightarrow & A \\
\downarrow \llbracket \cdot \rrbracket^0 & & \downarrow \llbracket \cdot \rrbracket \\
K(D) \cong_{\text{dual}} C(K\Omega(D)) & \hookrightarrow & K\Omega(D)
\end{array}$$

Remember that  $C(K\Omega(D))$  are just those compact-open subsets which are of the form  $\uparrow c$  for  $c \in K(D)$ . The inclusion order between such principal filters is dual to the usual order on  $K(D)$ .

Let us now lift the pre-isomorphism to the domain level. In the previous chapters, the natural approach would have been to apply the ideal completion functor to the pre-isomorphism between  $C(A)^{op}$  and  $K(D)$ . Here we use Stone-duality and apply  $\text{spec}$  to  $\llbracket \cdot \rrbracket$ . This yields an isomorphism between  $\text{spec}(A)$  and  $\text{spec}(K\Omega(D))$ . Composed with the inverse of the unit  $\eta$  it gives us the isomorphism  $\tau: \text{spec}(A) \rightarrow D$ .

$$\begin{array}{ccc}
\text{spec}(A) & & \\
\downarrow \text{spec}(\llbracket \cdot \rrbracket)^{-1} & \searrow \tau & \\
\text{spec}(K\Omega(D)) & \xrightarrow{\eta^{-1}} & D
\end{array}$$

It will be good to have a concrete idea of the behaviour of  $\tau$ , at least for compact elements of  $\text{spec}(A)$ . These are filters in  $A$  which are generated by  $\vee$ -prime elements. So let  $F = \uparrow a$  with  $a \in C(A)$ . It is easily checked that  $\tau(F)$  equals that compact element  $c$  of  $D$  which is least in the compact-open subset  $\llbracket a \rrbracket^0$ .

**Proposition 7.3.3.** *There exists a map  $\llbracket \cdot \rrbracket: A \rightarrow K\Omega(D)$  such that the domain prelocale  $A$  is a localic description of the bifinite domain  $D$  if and only if  $\text{spec}(A)$  and  $D$  are isomorphic.*

**Proof.** We have just described how to derive an isomorphism from a pre-isomorphism. For the converse observe that the unit  $\varepsilon: A \rightarrow K\Omega(\text{spec}(A))$  is surjective, order-preserving and order-reflecting (Proposition 7.2.4). ■

For more general functions between domains, we can translate join-approximable relations into the language of domain prelocales. The following is then just a slight extension of Theorem 7.2.30.

**Theorem 7.3.4.** *The category of domain prelocales and join-approximable relations is dually equivalent to the category of bifinite domains and Scott-continuous functions.*

Our attempt to mimic the cartesian product construction forced us to pass to preordered structures but once we have accepted this we can go

one step farther and make the prelocales syntactic objects in which no identifications are made at all. More precisely, it is no loss of generality to assume that the underlying algebra is a term algebra with respect to the operations  $\vee, \wedge, 0$ , and  $1$ . As an example, let us describe the one-point domain  $\mathbb{I}$  in this fashion. We take the term algebra on no generators, that is, every term is a combination of  $0$ s and  $1$ s. The preorder is the smallest relation compatible with the requirements in Definition 7.3.1. The effect of this is that there are exactly two equivalence classes with respect to  $\approx$ , the terms equivalent to  $1$  and the terms equivalent to  $0$ . The former are precisely the  $\vee$ -prime terms. We denote the resulting domain prelocale by  $\mathbb{I}$ .

The syntactic approach also suggests that we look at the following relation between domain prelocales:

**Definition 7.3.5.** Let  $A$  and  $B$  be domain prelocales. We say that  $A$  is a *sub-prelocale* of  $B$  if the following conditions are satisfied:

1.  $A$  is a subalgebra of  $B$  with respect to  $\vee, \wedge, 0$  and  $1$ .
2. The preorder on  $A$  is the restriction of the preorder on  $B$  to  $A$ .
3.  $C(A)$  equals  $A \cap C(B)$ .

We write  $A \trianglelefteq B$  if  $A$  is a sub-prelocale of  $B$ .

**Proposition 7.3.6.** If  $A$  is a sub-prelocale of  $B$  then the following defines an embedding projection pair between  $\text{spec}(A)$  and  $\text{spec}(B)$ :

$$\begin{aligned} e: \text{spec}(A) &\rightarrow \text{spec}(B), & e(F) &= \uparrow_B(F); \\ p: \text{spec}(B) &\rightarrow \text{spec}(A), & p(F) &= F \cap A. \end{aligned}$$

**Proof.** It is clear that both  $e$  and  $p$  are continuous because directed joins of elements in  $\text{spec}(A)$ , resp.  $\text{spec}(B)$ , are just directed unions of prime filters. We have  $p \circ e = \text{id}$  because the preorder on  $A$  is the restriction of that on  $B$ . For  $e \circ p \sqsubseteq \text{id}$  we don't need any special assumptions.

The crucial point is that the two functions are well-defined in the sense that they indeed produce prime filters. The filter part follows again from the fact that both operations and preorder on  $A$  are the restrictions of those on  $B$ . For primeness assume that  $\bigvee M \in \uparrow_B(F)$  for some finite  $M \subseteq B$ . This means  $x \lesssim \bigvee M$  for some  $x \in F$ . This element itself is a supremum of  $\vee$ -primes of  $A$  and because  $F$  is a prime filter in  $A$  we have some  $\vee$ -prime element  $x'$  below  $\bigvee M$  in  $F$ . But we have also required that the  $\vee$ -prime elements of  $A$  are precisely those  $\vee$ -prime elements of  $B$  which lie in  $A$  and therefore some  $m \in M$  must be above  $x'$ .

Primeness of  $F \cap A$ , on the other hand, follows easily because suprema in  $A$  are also suprema in  $B$ . ■

**Corollary 7.3.7.** Assume that  $A$  is a localic description of  $D$  via  $\llbracket \cdot \rrbracket_A$ , that  $B$  describes  $E$  via  $\llbracket \cdot \rrbracket_B$ , and that  $A \trianglelefteq B$ . Then the following defines

an embedding  $e$  of  $D$  into  $E$ :

If  $c \in K(D)$ ,  $a \in C(A)$ ,  $\llbracket a \rrbracket_A^0 = \uparrow c$ ,  $\llbracket a \rrbracket_B^0 = \uparrow d$ , then  $e(c) = d$ .

**Proof.** If we denote by  $e'$  the embedding from  $\text{spec}(A)$  into  $\text{spec}(B)$  as defined in the preceding proposition, then the embedding  $e: D \rightarrow E$  is nothing else but  $\tau_B \circ e' \circ \tau_A^{-1}$ . ■

Of course, it happens more often that  $\text{spec}(A)$  is a sub-domain of  $\text{spec}(B)$  than that  $A$  is a sub-prelocale of  $B$  but the fact is that it will be fully sufficient and even advantageous to work with the stronger relation when it comes to solving recursive domain equations.

### 7.3.2 Constructions: the general technique

Before we demonstrate how function space and Plotkin powerdomain can be constructed through prelocales, let us outline the general technique. The overall picture is in the following diagram. We explain how to get its ingredients step by step below.

$$\begin{array}{ccc}
 C(T(A, A')) & \hookrightarrow & T(A, A') \\
 \llbracket \cdot \rrbracket^0 \downarrow & & \downarrow \llbracket \cdot \rrbracket \\
 K(F_T(D, D')) & \cong_{\text{dual}} C(K\Omega(F_T(D, D'))) \hookrightarrow & K\Omega(F_T(D, D'))
 \end{array}$$

**1. The set-up.** We want to study a construction  $T$  on (bifinite) domains. This could be any one from Table 1 in Section 3.2.6 or a bilimit or one of the powerdomain constructions from Section 6.2. The diagram illustrates a binary construction. We can assume that we understand the action of the associated functor  $F_T$  on bifinite domains. In particular, we know what the compact elements of  $F_T(D, D')$  are, how they compare and how  $F_T$  acts on embeddings (Proposition 5.2.6). Thus we should have a clear understanding of the bottom row of the diagram, in detail:

- $F_T(D, D')$  is the effect of the functor  $F_T$  on objects  $D$  and  $D'$ .
- $K(F_T(D, D'))$  are the compact elements of  $F_T(D, D')$ .
- $K\Omega(F_T(D, D'))$  are the compact-open subsets of  $F_T(D, D')$  and these are precisely those upper sets which are of the form  $\uparrow u$  for a finite set  $u$  of compact elements.
- $C(K\Omega(F_T(D, D')))$  are the  $\vee$ -prime elements of  $K\Omega(F_T(D, D'))$  and these are precisely those subsets of  $F_T(D, D')$  which are of the form  $\uparrow c$  for  $c$  a compact element. The order is inclusion which is dual to the usual order on compact elements.

Furthermore, we assume that we are given domain prelocales  $A$  and  $A'$  which describe the bifinite domains  $D$  and  $D'$ , respectively. These descrip-



tions are encoded in pre-isomorphisms  $\llbracket \cdot \rrbracket_A: A \rightarrow K\Omega(D)$  and  $\llbracket \cdot \rrbracket_{A'}: A' \rightarrow K\Omega(D')$ .

**2. The goal.** We want to define a domain prelocale  $T(A, A')$  which is a localic description of  $F_T(D, D')$ . This is achieved in the following series of steps.

**3. Definition of  $T(A, A')$ .** This is the creative part of the enterprise. We search for a description of compact-open subsets of  $F_T(D, D')$  based on our knowledge of the compact-open subsets of  $D$  and  $D'$ . The point is to do this directly, *not* via the compact elements of  $D$ ,  $D'$ , and  $F_T(D, D')$ . There will be an immediate payoff, as we will gain an understanding of the construction in terms of properties rather than points. Our treatment of the Plotkin powerdomain below illustrates this most convincingly.

The definition of  $T(A, A')$  will proceed uniformly in all concrete instances. First a set  $G_T$  of generators is defined and then  $T(A, A')$  is taken to be the term algebra over  $G_T$  with respect to  $\vee, \wedge, 0$ , and  $1$ . An interpretation function  $\llbracket \cdot \rrbracket: G_T \rightarrow K\Omega(F_T(D, D'))$  is defined based on the interpretations  $\llbracket \cdot \rrbracket_A$  and  $\llbracket \cdot \rrbracket_{A'}$ . It is extended to all of  $T(A, A')$  as a lattice homomorphism:  $\llbracket a \vee b \rrbracket = \llbracket a \rrbracket \cup \llbracket b \rrbracket$ , etc. Finally, axioms and rules are given which govern the preorder and  $\vee$ -primeness predicate.

Next we have to check that our definitions work. This task is also broken into a series of steps as follows.

**4. Soundness.** We check that axioms and rules translate via  $\llbracket \cdot \rrbracket$  into valid statements about compact-open subsets of  $F_T(D, D')$ . This is usually quite easy. From soundness we infer that  $\llbracket \cdot \rrbracket$  is monotone and can be restricted and corestricted to a map  $\llbracket \cdot \rrbracket^0: C(T(A, A')) \rightarrow C(K\Omega(F_T(D, D')))$ .

**5. Prime generation.** Using the axioms and rules, we prove that every element of  $T(A, A')$  can be transformed (effectively) into an equivalent term which is a finite supremum of expressions which are asserted to be  $\vee$ -prime. This is the crucial step and usually contains the main technical work. It allows us to prove the remaining properties of  $\llbracket \cdot \rrbracket$  through  $\llbracket \cdot \rrbracket^0$  and for the latter we can use our knowledge of the basis of  $F_T(D, D')$ .

**6. Completeness for  $\vee$ -primes.** We show that  $\llbracket \cdot \rrbracket^0$  is order reflecting.

**7. Definability for  $\vee$ -primes.** We show that  $\llbracket \cdot \rrbracket^0$  is surjective.

At this point we can fill in the remaining pieces without reference to the concrete construction under consideration.

**8. Completeness.** The interpretation function  $\llbracket \cdot \rrbracket$  itself is order-reflecting.

**Proof.** Let  $a, b \in T(A, A')$  be such that  $\llbracket a \rrbracket \subseteq \llbracket b \rrbracket$ . By 5 we can replace these expressions by formal joins of  $\vee$ -primes:  $a \approx a_1 \vee \cdots \vee a_n$  and  $b \approx b_1 \vee \cdots \vee b_m$ . Soundness ensures that the value under the interpretation function remains unchanged and that each  $\llbracket a_i \rrbracket$  (resp.  $\llbracket b_j \rrbracket$ ) is of the form  $\uparrow c_i$  (resp.  $\uparrow d_j$ ) for  $c_i, d_j$  compact elements in  $F_T(D, D')$ . The inclusion order on  $K\Omega(F_T(D, D'))$  translates into the formula  $\forall i \exists j. \uparrow c_i \subseteq \uparrow d_j$  which by the

completeness for  $\vee$ -primes can be pulled back into  $T(A, A')$ :  $\forall i \exists j. a_i \lesssim b_j$ . In every preordered lattice it must follow that  $a \lesssim b$  holds. ■

**9. Definability.** The surjectivity of  $\llbracket \cdot \rrbracket$  is an easy consequence of the surjectivity of  $\llbracket \cdot \rrbracket^0$  because we know that compact-open subsets in an algebraic domain are finite unions of compactly generated principal filters.

**10. Well-definedness.** Of course,  $\mathsf{K}\Omega(F_T(D, D'))$  is a domain prelocale and we have just shown that pre-order and primeness predicate on  $T(A, A')$  are preserved and reflected by  $\llbracket \cdot \rrbracket$ . This constitutes a semantic proof that  $T(A, A')$  satisfies the two extra conditions for domain prelocales. In other words,  $T$  is a well-defined operation on domain prelocales.

**11. Stone-duality.** At this point we have shown that  $\llbracket \cdot \rrbracket$  is a pre-isomorphism. As in the previous subsection we lift it to an isomorphism  $\tau$  between  $\mathsf{spec}(T(A, A'))$  and  $F_T(D, D')$  via Stone duality:

$$\begin{array}{ccc}
 \mathsf{spec}(T(A, A')) & & \\
 \downarrow \mathsf{spec}(\llbracket \cdot \rrbracket)^{-1} & \searrow \tau & \\
 \mathsf{spec}(\mathsf{K}\Omega(F_T(D, D'))) & \xrightarrow{\eta^{-1}} & F_T(D, D')
 \end{array}$$

So much for the correspondence on the object level. We also want to see how the construction  $T$  harmonizes with the sub-prelocale relation, on the one hand, and the isomorphism  $\tau$ , on the other hand. Thus we assume that we are given two more prelocales,  $B$  and  $B'$ , which are localic descriptions of bifinite domains  $E$  and  $E'$ , such that  $A \trianglelefteq B$  and  $A' \trianglelefteq B'$  hold. In Corollary 7.3.7 we have seen how to define from this embeddings  $e: D \rightarrow E$  and  $e': D' \rightarrow E'$ . In Proposition 5.2.6 we have shown how the functors associated with different constructions act on embeddings, hence we may unambiguously write  $F_T(e, e')$  for the result of this action, which is an embedding from  $F_T(D, D')$  to  $F_T(E, E')$ . Embeddings preserve compact elements so  $F_T(e, e')$  restricts and corestricts to a monotone function  $F_T(e, e')^0: \mathsf{K}(F_T(D, D')) \rightarrow \mathsf{K}(F_T(E, E'))$ . Now for both  $T(A, A')$  and  $T(B, B')$  we have a diagram such as depicted at the beginning of this subsection. We connect the lower left corners of these by  $F_T(e, e')^0$ . This gives rise also to a map  $i$  from  $\mathsf{C}(\mathsf{K}\Omega(F_T(D, D')))$  to  $\mathsf{C}(\mathsf{K}\Omega(F_T(E, E')))$ . Our way of defining  $T(A, A')$  will be such that it is immediate that  $\mathsf{C}(T(A, A'))$  is a subset of  $\mathsf{C}(T(B, B'))$  and hence there is an inclusion map connecting the upper left corners. Our next technical step then is the following.

**12. Naturality.** We show that the diagram

$$\begin{array}{ccc}
C(T(A, A')) & \hookrightarrow & C(T(B, B')) \\
\downarrow \llbracket \cdot \rrbracket_{T(A, A')}^0 & & \downarrow \llbracket \cdot \rrbracket_{T(B, B')}^0 \\
C(K\Omega(F_T(D, D'))) & \xrightarrow{i} & C(K\Omega(F_T(E, E')))
\end{array}$$

commutes. On the element level this reads: If  $a \in C(T(A, A'))$  and  $\llbracket a \rrbracket_{T(A, A')}^0 = \uparrow c$  and  $\llbracket a \rrbracket_{T(B, B')}^0 = \uparrow d$  then  $F_T(e, e')^0(c) = d$ . Now we can again get the remaining missing information in a general manner.

**13. Monotonicity.** We show that  $T(A, A') \trianglelefteq T(B, B')$ . From the form of our construction it will be clear that  $T(A, A')$  is a subset of  $T(B, B')$  and the axioms and rules will be such that whatever can be derived in  $T(A, A')$  can also be derived in  $T(B, B')$ . We must show that in the larger prelocale nothing extra can be proved for elements of  $T(A, A')$ . The argument is a semantic one.

**Proof.** Let  $a, a' \in C(T(A, A'))$  such that  $a \lesssim a'$  holds in  $T(B, B')$ . Let  $\llbracket a \rrbracket_{T(A, A')}^0 = \uparrow c$ ,  $\llbracket a \rrbracket_{T(B, B')}^0 = \uparrow d$  and similarly for  $a'$ . Correctness says that  $\uparrow d \subseteq \uparrow d'$  and hence  $d \sqsupseteq d'$ . By naturality we have  $F_T(e, e')^0(c) = d \sqsupseteq d' = F_T(e, e')^0(c')$ . Embeddings are order reflecting so  $c \sqsupseteq c'$  follows. Completeness then allows us to conclude that  $a \lesssim a'$  holds in  $T(A, A')$  as well.

In the same way it is seen that the predicate  $C$  on  $T(A, A')$  is the restriction of that on  $T(B, B')$ . ■

**14. Least prelocale.** It follows from the correctness of the construction that  $1 \trianglelefteq T(A, A')$  holds.

**15. Naturality of  $\tau$ .** Having established the relation  $T(A, A') \trianglelefteq T(B, B')$  we can look at the embedding  $I: \text{spec}(T(A, A')) \rightarrow \text{spec}(T(B, B'))$  which we defined in Proposition 7.3.6. We claim that the following diagram commutes:

$$\begin{array}{ccc}
\text{spec}(T(A, A')) & \xrightarrow{I} & \text{spec}(T(B, B')) \\
\downarrow \tau_A & & \downarrow \tau_B \\
F_T(D, D') & \xrightarrow{F_T(e, e')} & F_T(E, E')
\end{array}$$

In other words,  $F_T(e, e')$  equals the embedding which can be derived from  $T(A, A') \trianglelefteq T(B, B')$  in the general manner of Corollary 7.3.7.

**Proof.** This is a diagram of bifinite domains and Scott-continuous functions. It therefore suffices to check commutativity for compact elements. A compact element in  $\text{spec}(T(A, A'))$  is a filter  $F$  generated by a term

$a \in C(T(A, A'))$ . Its image under  $\tau_A$  is the compact element  $c$  which generates the compact-open subset  $\llbracket a \rrbracket_{T(A, A')}^0$ . The filter  $I(F)$  is generated by the same term  $a$ . Applying  $\tau_B$  to it gives us a compact element  $d$  which is least in  $\llbracket a \rrbracket_{T(A, A')}^0$ . Step 12 ensures that  $F_T(e, e')$  maps  $c$  to  $d$ . ■

### 7.3.3 The function space construction

We start out with two preparatory lemmas. The following notation will be helpful. We write  $(A \Rightarrow B)$  for the set of functions which map all of  $A$  into  $B$ .

**Lemma 7.3.8.** *The Scott-topology on the function space  $[D \rightarrow D']$  for bifinite domains  $D$  and  $D'$  equals the compact-open topology.*

**Proof.** Let  $A \subseteq D$  be compact and  $O \subseteq D'$  be open and let  $F \subseteq [D \rightarrow D']$  be a directed set of continuous functions for which  $\bigcup^\uparrow F$  maps  $A$  into  $O$ . For every  $x \in A$  we have  $(\bigcup^\uparrow F)(x) \in O$  and because  $O$  is open, there is  $f_x \in F$  with  $f_x(x) \in O$ . The collection of open sets of the form  $f_x^{-1}(O)$ ,  $x \in A$ , covers  $A$ . By compactness, this is true for finitely many  $f_x^{-1}(O)$  already. If we let  $f$  be an upper bound in  $F$  for these  $f_x$ , then  $A \subseteq f^{-1}(O)$  holds which is equivalent to  $f(A) \subseteq O$ . Hence  $(A \Rightarrow O)$  is a Scott-open set in  $[D \rightarrow D']$ .

If, on the other hand,  $f$  belongs to a Scott-open open set  $O \subseteq [D \rightarrow D']$ , then this is true also for some approximation  $g'_m \circ f \circ g_n$  with  $g_n$  an idempotent deflation on  $D$ ,  $g'_m$  an idempotent deflation on  $D'$ . For each element  $x$  in the image of  $g_n$  we have the set  $(\uparrow x \Rightarrow (\uparrow g'_m \circ f \circ g_n(x)))$ . The intersection of all these belongs to the compact-open topology, contains  $f$ , and is contained in  $O$ . ■

**Lemma 7.3.9.** *Let  $D$  and  $D'$  be bifinite and let  $A \subseteq D$  and  $A' \subseteq D'$  be compact-open. Then  $(A \Rightarrow A')$  is compact-open in  $[D \rightarrow D']$ .*

**Proof.** We know that  $(A \Rightarrow A')$  defines an open set by the previous lemma. From bifiniteness we get idempotent deflations  $g_n$  on  $D$  and  $g'_m$  on  $D'$  such that  $A = \uparrow g_n(A)$  and  $A' = \uparrow g'_m(A')$ . It follows that  $(A \Rightarrow A') = \uparrow G_{nm}(A \Rightarrow A')$  for the idempotent deflation  $G_{nm}$  on  $[D \rightarrow D']$  which maps  $f$  to  $g'_m \circ f \circ g_n$ . ■

Now let  $A$  and  $A'$  be domain prelocales describing bifinite domains  $D$  and  $D'$ , as outlined in the general scheme in the previous subsection. The two lemmas justify the following choice of generators and interpretation function for our localic function space construction:

$$\begin{aligned} G_{\rightarrow} &= \{(a \rightarrow a') \mid a \in A, a' \in A'\}; \\ \llbracket (a \rightarrow a') \rrbracket &= (\llbracket a \rrbracket_A \Rightarrow \llbracket a' \rrbracket_{A'}) \end{aligned}$$

Note that the elements  $(a \rightarrow a')$  are just syntactic expressions. Here are axioms and rules for the pre-order and C-predicate.



**Axioms**

$$\begin{aligned}
(\rightarrow - \wedge) \quad & (a \rightarrow \bigwedge_{i \in I} a'_i) \approx \bigwedge_{i \in I} (a \rightarrow a'_i). \\
(\rightarrow - \vee - l) \quad & (\bigvee_{i \in I} a_i \rightarrow a') \approx \bigwedge_{i \in I} (a_i \rightarrow a'). \\
(\text{dist}) \quad & a \wedge (b \vee c) \approx (a \wedge b) \vee (a \wedge c).
\end{aligned}$$

**Rules**

$$\begin{aligned}
(\rightarrow - \vee - r) \quad & \text{If } C(a) \text{ then } (a \rightarrow \bigvee_{i \in I} a'_i) \approx \bigvee_{i \in I} (a \rightarrow a'_i). \\
(\rightarrow - \lesssim) \quad & \text{If } b \lesssim a \text{ and } a' \lesssim b' \text{ then } (a \rightarrow a') \lesssim (b \rightarrow b'). \\
(\rightarrow - C) \quad & \text{If } \forall i \in I. (C(a_i) \text{ and } C(a'_i)) \text{ and if } \forall K \subseteq I \exists L \subseteq I. \\
& (\bigwedge_{k \in K} a_k \approx \bigvee_{l \in L} a_l \text{ and } (\forall k \in K, l \in L. a'_k \lesssim a'_l)) \\
& \text{then } C(\bigwedge_{i \in I} (a_i \rightarrow a'_i)).
\end{aligned}$$

A few comments about these formulae are in place. First a convention: we assume that all index sets are finite, so that the expressions  $\bigwedge_{i \in I} a_i$ , etc., do indeed belong to the term algebra over  $G_{\perp}$ . Observe the use of the C-predicate in the rule  $(\rightarrow - \vee - r)$ . Without it, it would be very difficult to express this property. Also note that we enforce distributivity. This will be a prerequisite to prove prime generation below.

It is clear that the rules are sound for the given interpretation, in particular,  $(\rightarrow - C)$  is the exact mirror image of our definition of joinable families of step functions, Definition 4.2.2. Let us therefore immediately turn to the crucial step 5. We cannot use Lemma 7.3.9 directly because we have not encoded the idempotent deflations. We must find the minimal elements of a compact-open subset explicitly. We illustrate the general technique in an example.

Suppose  $\llbracket a \rrbracket_A$  is of the form  $\uparrow c \cup \uparrow d$  and  $\llbracket a' \rrbracket_{A'}$  is of the form  $\uparrow c' \cup \uparrow d'$ . We get a minimal element of  $((\uparrow c \cup \uparrow d) \Rightarrow (\uparrow c' \cup \uparrow d'))$  by choosing a value  $f(c)$  and a value  $f(d)$  from  $\{c', d'\}$ . Then we must look at the intersection  $\uparrow c \cap \uparrow d$  which again is of the form  $\uparrow e_1 \cup \dots \cup \uparrow e_n$  by coherence. For each  $e_i$  we must choose a value from  $\text{mub}\{f(c), f(d)\} = \{e'_1, \dots, e'_m\}$ . And so on. Bifiniteness of the argument domain ensures that this process stops after a finite number of iterations and that the result is a joinable family of pairs  $\langle x, f(x) \rangle$ . Coherence of the result domain guarantees that all in all only finitely many choices are possible. (Note that it can happen that a set of minimal upper bounds in the image domain is empty. In this case we have just been unlucky with our choices. If  $\llbracket a' \rrbracket_{A'}$  is not empty then some minimal function exists.)

We can mimic this procedure in the prelocale as follows. For simplicity and to make the analogy apparent, we let  $c, d$  stand for terms such that  $C(c), C(d)$  and  $a \approx c \vee d$ . Similarly for  $a'$ . We get:



$$\begin{aligned}
& (a \rightarrow a') \approx \\
& \approx ((c \vee d) \rightarrow (c' \vee d')) & (\rightarrow - \lesssim) \\
& \approx (c \rightarrow (c' \vee d')) \wedge (d \rightarrow (c' \vee d')) & (\rightarrow - \vee - l) \\
& \approx ((c \rightarrow c') \vee (c \rightarrow d')) \wedge ((d \rightarrow c') \vee (d \rightarrow d')) & (\rightarrow - \vee - r) \\
& \approx ((c \rightarrow c') \wedge (d \rightarrow d')) \vee \dots (3 \text{ more terms}) & (\text{dist})
\end{aligned}$$

We follow up only the first of these four terms. The trick is to smuggle in the  $\vee$ -prime terms  $e_1, \dots, e_n$  whose join equals  $c \wedge d$ .

$$\begin{aligned}
& (c \rightarrow c') \wedge (d \rightarrow d') \approx \\
& \approx ((c \vee e_1 \vee \dots \vee e_n) \rightarrow c') \wedge ((d \vee e_1 \vee \dots \vee e_n) \rightarrow d') & (\rightarrow - \lesssim) \\
& \approx (c \rightarrow c') \wedge (d \rightarrow d') \wedge ((e_1 \vee \dots \vee e_n) \rightarrow (c' \wedge d')) & (\rightarrow - \vee - l) \\
& \approx (c \rightarrow c') \wedge (d \rightarrow d') \wedge ((e_1 \vee \dots \vee e_n) \rightarrow (e'_1 \vee \dots \vee e'_m))
\end{aligned}$$

and now induction may do its job. Eventually we will have transformed  $(a \rightarrow a')$  into a disjunction of joinable families. For these,  $\vee$ -primeness may be inferred through rule  $(\rightarrow - C)$ . Note that distributivity allows us to replace every term by an equivalent term of the form  $\bigvee (\bigwedge (a_i \rightarrow a'_i))$  and for each term of the form  $\bigwedge (a_i \rightarrow a'_i)$  the transformation works as illustrated.

Next we show completeness for  $\vee$ -primes. So assume  $a$  and  $b$  are terms for which the  $C$ -predicate holds and for which  $\llbracket a \rrbracket \subseteq \llbracket b \rrbracket$ . It must be the case that  $a$  and  $b$  are equivalent to joinable families  $\bigwedge_{i \in I} (a_i \rightarrow a'_i)$  and  $\bigwedge_{j \in J} (b_j \rightarrow b'_j)$  as there is no other way of deriving  $\vee$ -primeness in  $[A \rightarrow A']$ . The order relation between joinable families has been characterized in Lemma 4.2.3. Here it says:  $\forall i \in I \exists j \in J. (\llbracket b_j \rrbracket \subseteq \llbracket a_i \rrbracket \text{ and } \llbracket a'_i \rrbracket \subseteq \llbracket b'_j \rrbracket)$ . Since we assume completeness for the constituting prelocales  $A$  and  $A'$ , we may infer  $\forall i \in I \exists j \in J. (b_j \lesssim a_i \text{ and } a'_i \lesssim b'_j)$ . The relation  $a \lesssim b$  is now easily derived from  $(\rightarrow - \lesssim)$ .

Definability for  $\vee$ -primes is immediate because we know that all compact functions arise from joinable families (Lemma 4.2.3 and Proposition 4.2.4).

Properties 8 to 11 follow for all constructions uniformly. We are left with proving naturality, Property 12. To this end, let us first see how the embedding  $[e \rightarrow e']$  transforms a step function  $(a \searrow a')$ . We have:  $[e \rightarrow e']((a \searrow a')) = (a \searrow e'(a')) \circ e^*$  and  $(a \searrow e'(a')) \circ e^*(x) = e'(a') \iff a \sqsubseteq e^*(x) \iff e(a) \sqsubseteq x$ . We get the step function  $(e(a) \searrow e'(a'))$ .

Now let  $a \approx \bigwedge_{i \in I} (a_i \rightarrow a'_i)$  be an element of  $[A \rightarrow A']$  for which  $C(a)$  holds. The interpretation  $\llbracket a \rrbracket_{[A \rightarrow A']}^0$  of  $a$  is the upper set generated by the joinable family of step functions  $(c_i \searrow c'_i)$ , where  $\llbracket a_i \rrbracket_A^0 = \uparrow c_i$  and  $\llbracket a'_i \rrbracket_{A'}^0 = \uparrow c'_i$  for all  $i \in I$ . Applying the embedding  $[e \rightarrow e']$  to these gives us the step functions  $(e(c_i) \searrow e'(c'_i))$  as we have just seen. By Corollary 7.3.7 we can rewrite these as  $(d_i \searrow d'_i)$ , where  $\llbracket a_i \rrbracket_B^0 = \uparrow d_i$  and

$\llbracket a'_i \rrbracket_{B'}^0 = \uparrow d'_i$ . The supremum of the joinable family  $((d_i \searrow d'_i))_{i \in I}$  is least in  $\llbracket a \rrbracket_{[B \rightarrow B']}^0$ . This was to be proved.

Taking  $D$  to be  $\text{spec}(A)$  and  $E$  to be  $\text{spec}(B)$  we can express the faithfulness of our localic construction quite concisely as follows:

**Theorem 7.3.10.** *Let  $A$  and  $B$  be domain prelocales. Then*

$$[\text{spec}(A) \rightarrow \text{spec}(B)] \cong \text{spec}([A \rightarrow B])$$

*and this isomorphism is natural with respect to the sub-prelocale relation.*

### 7.3.4 The Plotkin powerlocale

Next we want to describe the lattice of compact-open subsets of the Plotkin powerdomain of a bifinite domain  $D$ . By Theorem 6.2.22 we know that  $P^P(D)$  is concretely represented as the set of lenses in  $D$ , ordered by the Egli–Milner ordering (Definition 6.2.2). The compact elements in  $P^P(D)$  are those lenses which are convex closures of finite non-empty subsets of  $K(D)$  (Proposition 6.2.6). Idempotent deflations  $d$  on  $D$  can be lifted to  $P^P(D)$  because  $P^P$  is a functor. They map a lens  $L$  to the convex closure of  $d(L)$ .

The compact-open subsets of  $P^P(D)$ , however, are not so readily described. The problem is that one half of the Egli–Milner ordering refers to closed lower sets rather than upper sets. We do not follow this up as there is no logical pathway from the order theory to the axiomatization we are aiming for. It is much more efficient either to consult the mathematical literature on hyperspaces (see [Vietoris, 1921; Vietoris, 1922; Smyth, 1983b]) or to remind ourselves that powerdomains were introduced to model non-deterministic behaviour. If we think of the compact-open subsets in  $D$  as observations that can be made about outcomes of a computation, then it is pretty clear that there are two ways of using these to make statements about non-deterministic programs: it could be the case that all runs of the program satisfy the property or it could be that at least one run satisfies it. Let us check the mathematics:

**Lemma 7.3.11.** *If  $D$  is a bifinite domain and  $O$  is compact-open in  $D$ , then the following are compact-open subsets in  $P^P(D)$ :*

$$\begin{aligned} A(O) &= \{L \in \text{Lens}(D) \mid L \subseteq O\}, \\ E(O) &= \{L \in \text{Lens}(D) \mid L \cap O \neq \emptyset\}, \end{aligned}$$

*Furthermore, if we let  $O$  range over all compact-open subsets in  $D$  then the collection of all  $A(O)$  and  $E(O)$  forms a base for the Scott-topology on  $P^P(D)$ .*

**Proof.** Let  $O$  be compact-open. Then  $O$  is the upper set of finitely many compact elements and we find an idempotent deflation  $d$  such that  $O =$

$\uparrow d(O)$ . It is clear that for  $\hat{d} = P^P(d)$  we have both  $A(O) = \uparrow \hat{d}(A(O))$  and  $E(O) = \uparrow \hat{d}(E(O))$ . Hence these sets are compact-open, too.

Let  $K$  be a compact lens, that is, of the form  $C \times (u)$  for  $u \subseteq_{\text{fin}} K(D)$ . The upper set of  $K$  in  $P^P(D)$  can be written as  $A(\uparrow u) \cap \bigcap_{c \in u} E(\uparrow c)$ . ■

The following definition then comes as no surprise:

**Definition 7.3.12.** Let  $A$  be a domain prelocale which is a localic description of the bifinite domain  $D$ . We define the *Plotkin powerlocale*  $P^P(A)$  over  $A$  as the term algebra over the generators

$$G_P = \{\Box a \mid a \in A\} \cup \{\Diamond a \mid a \in A\}$$

with the interpretation function  $\llbracket \cdot \rrbracket : P^P(A) \rightarrow K\Omega(P^P(D))$  defined by

$$\llbracket \Box a \rrbracket = A(\llbracket a \rrbracket), \quad \llbracket \Diamond a \rrbracket = E(\llbracket a \rrbracket)$$

on the generators and extended to  $P^P(A)$  as a lattice homomorphism.

Pre-order and  $C$ -predicate are defined as follows:

#### Axioms

- $(\Box - \wedge)$   $\Box(\bigwedge_{i \in I} a_i) \approx \bigwedge_{i \in I} \Box a_i,$
- $(\Box - 0)$   $\Box 0 \approx 0,$
- $(\Diamond - \vee)$   $\Diamond(\bigvee_{i \in I} a_i) \approx \bigvee_{i \in I} \Diamond a_i,$
- $(\Diamond - 1)$   $\Diamond 1 \approx 1,$
- $(\Box - \vee)$   $\Box(a \vee b) \lesssim \Box a \vee \Diamond b,$
- $(\Diamond - \wedge)$   $\Diamond a \wedge \Diamond b \lesssim \Diamond(a \wedge b),$
- (dist)  $a \wedge (b \vee c) \approx (a \wedge b) \vee (a \wedge c).$

#### Rules

- $(P - \lesssim)$  If  $a \lesssim b$  then  $\Box a \lesssim \Box b$  and  $\Diamond a \lesssim \Diamond b,$
- $(P - C)$  If  $C(a_i)$  holds for all  $i \in I$  and  $I$  is non-empty, then  
 $C(\Box(\bigvee_{i \in I} a_i) \wedge \bigwedge_{i \in I} \Diamond a_i).$

Note that we again require distributivity explicitly. The derivation scheme is almost minimal (in combination with the rest,  $(\Box - 0)$  and  $(\Diamond - 1)$  are equivalent). The following derived axioms are more useful than  $(\Box - \vee)$  and  $(\Diamond - \wedge)$ :

- (D1)  $\Box(a \vee b) \approx \Box a \vee (\Box(a \vee b) \wedge \Diamond b),$
- (D2)  $\Box a \wedge \Diamond b \approx \Box a \wedge \Diamond(a \wedge b).$

We leave it to the interested reader to check soundness and pass straight on to the central Step 5, which is generation by  $\vee$ -prime elements.

**Proof.** Given an expression in  $P^P(A)$  we first transform it into a disjunction of conjunctions by using the distributivity axiom. Thus it suffices to represent a term of the form

$$\bigwedge_{i \in I} \Box a_i \wedge \bigwedge_{j \in J} \Diamond b_j$$

as a disjunction of  $\vee$ -primes. But we can simplify further. Using  $(\Box - \wedge)$  we can pack all  $\Box$ -generators into a single term  $\Box a$  and by (D2) we can assume that for each  $j \in J$  we have  $b_j \lesssim a$ . We represent each  $b_j$  as a disjunction of  $\vee$ -primes of  $A$  and applying  $(\Diamond - \vee)$  and distributivity again we arrive at a disjunction of terms of the form

$$\Box a \wedge \bigwedge_{j=1}^m \Diamond d_j$$

where each  $d_j \in C(A)$ . Now we write  $a$  as a disjunction of  $\vee$ -primes  $c_i$ . Since each  $d_j$  is below  $a$ , it doesn't hurt to add these, too. We obtain

$$\Box(c_1 \vee \dots \vee c_n \vee d_1 \vee \dots \vee d_m) \wedge \bigwedge_{j=1}^m \Diamond d_j.$$

As yet we can not apply the  $\vee$ -primeness rule  $(P - C)$  because the two sets  $\{c_1, \dots, c_n, d_1, \dots, d_m\}$  and  $\{d_1, \dots, d_m\}$  may fail to coincide. Looking at the semantics for a moment, we see that in the compact-open subset thus described the minimal lenses are (the convex closures of) the least elements from each  $\llbracket d_j \rrbracket_A^0$  plus some of the generators of the  $\llbracket c_i \rrbracket_A^0$ . We therefore take our term further apart so as to have a  $\vee$ -prime expression for each subset of  $\{c_1, \dots, c_n\}$ . For this we use (D1). One application (plus some distributivity) yields

$$\begin{aligned} & (\Box(c_2 \vee \dots \vee c_n \vee d_1 \vee \dots \vee d_m) \wedge \bigwedge_{j=1}^m \Diamond d_j) \vee \\ & (\Box(c_1 \vee \dots \vee c_n \vee d_1 \vee \dots \vee d_m) \wedge \Diamond c_1 \wedge \bigwedge_{j=1}^m \Diamond d_j) \end{aligned}$$

and the picture becomes obvious. ■

Next we check that  $\llbracket \cdot \rrbracket^0$  is order-reflecting.

**Proof.** Assume  $\llbracket \Box(\bigvee_{i \in I} a_i) \wedge \bigwedge_{i \in I} \Diamond a_i \rrbracket^0 \subseteq \llbracket \Box(\bigvee_{i \in I} b_i) \wedge \bigwedge_{j \in J} \Diamond b_j \rrbracket^0$  and let  $c_i$  and  $d_j$  be the least compact elements in  $\llbracket a_i \rrbracket_A^0$ , respectively  $\llbracket b_j \rrbracket_A^0$ . Then we have  $\{d_j \mid j \in J\} \subseteq_{EM} \{c_i \mid i \in I\}$ , that is,

$$\begin{aligned} \forall i \in I \exists j \in J. \quad \uparrow c_i &\subseteq \uparrow d_j, \\ \forall j \in J \exists i \in I. \quad \uparrow c_i &\subseteq \uparrow d_j. \end{aligned}$$

Since we assume that  $\llbracket \cdot \rrbracket_A^0$  is order-reflecting, we get from the first equation  $\bigvee_{i \in I} a_i \lesssim \bigvee_{j \in J} b_j$  and from the second  $\bigwedge_{i \in I} \Diamond a_i \lesssim \bigwedge_{j \in J} \Diamond b_j$ . ■

The definability for  $\vee$ -primes has already been shown in Lemma 7.3.11. Hence we are left with checking naturality, which is Step 12.

**Proof.** Let  $t = \Box(\bigvee_{i \in I} a_i) \wedge \bigwedge_{i \in I} \Diamond a_i$  be a  $\vee$ -prime element in  $P^P(A)$  and let  $A$  be a sub-prelocale of  $B$ . Let  $e$  be the associated embedding from  $D$  to  $E$ . The least element in  $\llbracket t \rrbracket_{P^P(A)}^0$  is the convex closure of the set of minimal elements  $c_i$  in  $\llbracket a_i \rrbracket_A^0$ . Applying  $P^P(e)$  to it gives the convex closure of  $\{e(c_i) \mid i \in I\}$ , as we have argued in the remark following Theorem 6.1.9. Corollary 7.3.7 tells us that this is the least element in  $\llbracket t \rrbracket_{P^P(B)}^0$ . ■

As in the case of the function space construction we summarize:

**Theorem 7.3.13.** *Let  $A$  be a domain prelocale. Then*

$$P^P(\text{spec}(A)) \cong \text{spec}(P^P(A))$$

*and this isomorphism is natural with respect to the sub-prelocale relation.*

The prelocales for Hoare and Smyth powerdomain are much easier to describe. All we have to do is to elide all generators and rules which refer to  $\Box$ , respectively  $\Diamond$ .

### 7.3.5 Recursive domain equations

In this subsection we will treat bilimits in the same fashion as we have studied finitary constructions. We assume that we are given domain prelocales  $A_0 \trianglelefteq A_1 \trianglelefteq A_2 \trianglelefteq \dots$  such that each  $A_n$  describes some bifinite domain  $D_n$ . Corollary 7.3.7 states how the sub-prelocale relation between  $A_n$  and  $A_m$ , for  $n \leq m$ , translates into an embedding  $e_{mn}: D_n \rightarrow D_m$ . It is seen easily that  $\langle (D_n)_{n \in \mathbb{N}}, (e_{mn})_{n \leq m} \rangle$  is an expanding system, that is, for  $n \leq m \leq k$ ,  $e_{kn} = e_{km} \circ e_{mn}$  holds. We claim that the directed union  $A = \bigcup_{n \in \mathbb{N}} A_n$  is a domain prelocale which describes  $D = \text{bilim } D_n$ . The first claim is fairly obvious as all requirements about prelocales refer to finitely many elements only and hence a property of  $A$  can be inferred from its validity in some  $A_n$ . For the second claim we need to specify the interpretation function. To this end let  $l_m$  be the embedding of  $D_m$  into the bilimit (as defined in Theorem 3.3.7). Then we can set  $\llbracket a \rrbracket = l_m(\llbracket a \rrbracket_{A_m})$  where  $m \in \mathbb{N}$  is such that  $a$  is contained in  $A_m$ . The exact choice of  $m$  does not matter; if  $m \leq k$  then by Corollary 7.3.7 we have:  $\llbracket a \rrbracket_{A_k} = e_{km}(\llbracket a \rrbracket_{A_m})$  and applying  $l_k$  to this yields  $l_k(\llbracket a \rrbracket_{A_k}) = l_k \circ e_{km}(\llbracket a \rrbracket_{A_m}) = l_m(\llbracket a \rrbracket_{A_m})$ . The interpretation function is well-defined because embeddings preserve the order of approximation (Proposition 3.1.14), hence compact elements and compact-open subsets are also preserved.

In order to see that  $\llbracket \cdot \rrbracket$  is a pre-isomorphism we proceed as before, checking Steps 4, 5, 6, 7, and 12. It is, actually, rather simple. Soundness



holds because the  $l_m$  are monotone and map compact elements to compact elements. Prime generation holds because it holds in each  $A_m$ . Since the  $l_m$  are also order-reflecting we get completeness from the completeness of the  $[\![\cdot]\!]_{A_m}$ . Definability follows from Theorem 3.3.11; the only compact elements in  $D$  are the images (under  $l_n$ ) of compact elements in the approximating  $D_n$ . If we are given a second sequence  $B_0 \trianglelefteq B_1 \trianglelefteq B_2 \trianglelefteq \dots$  of prelocales (describing  $E_0, E_1, \dots$ ) such that for each  $n \in \mathbb{N}$  we have  $A_n \trianglelefteq B_n$  then it is clear that  $A \trianglelefteq B = \bigcup_{n \in \mathbb{N}} B_n$  holds, too. For naturality (Step 12) we must relate this to the embedding  $e$  from  $D$  to  $E = \text{bilim} E_n$ . The exact form of the latter can be extracted from Theorem 3.3.7:  $e = \bigsqcup_{n \in \mathbb{N}} k_n \circ e_n \circ l_n^*$ , where  $k_n$  is the embedding of  $E_n$  into  $E$  and  $e_n: D_n \rightarrow E_n$  is the embedding derived from  $A_n \trianglelefteq B_n$ . Now let  $a$  be  $\vee$ -prime in  $A$ . We have

$$\begin{aligned}
 e([\![a]\!]_A^0) &= \left( \bigsqcup_{n \in \mathbb{N}} k_n \circ e_n \circ l_n^* \right) (l_m([\![a]\!]_{A_m}^0)) \\
 &= \bigsqcup_{n \geq m} k_n \circ e_n([\![a]\!]_{A_m}^0) \\
 &= \bigsqcup_{n \geq m} k_n([\![a]\!]_{B_m}^0) \\
 &= [\![a]\!]_B^0,
 \end{aligned}$$

and our proof is complete.

**Theorem 7.3.14.** *If  $A_0 \trianglelefteq A_1 \trianglelefteq A_2 \trianglelefteq \dots$  is a chain of domain prelocales, then*

$$\text{spec}\left(\bigcup_{n \in \mathbb{N}} A_n\right) \cong \text{bilim}(\text{spec}(A_n))_{n \in \mathbb{N}}.$$

Observe how simple the limit operation for prelocales is if compared with a bilimit. This comes to full flower if we look at recursive domain equations. If  $T$  is a construction built from those which can be treated locally (we have seen function space, Plotkin powerdomain, and bilimit, but all the others from Section 3.2 can also be included) then we can find the initial fixpoint of the functor  $F_T$  on the localic side by simply taking the union of  $1 \trianglelefteq T(1) \trianglelefteq T(T(1)) \trianglelefteq \dots$ . Why does this work and why does the result describe the canonical fixpoint of  $F_T$ ? First of all, we have  $1 \trianglelefteq T(1)$  by Step 14. Successively applying  $T$  to this relation gives us  $T^n(1) \trianglelefteq T^{n+1}(1)$  by monotonicity (Step 13). Hence we do have a chain  $1 \trianglelefteq T(1) \trianglelefteq T(T(1)) \trianglelefteq \dots$  as stated, and we can form its union  $A$ . It obviously is a fixpoint of the construction  $T$  and therefore the domain  $D$  described by it is a fixpoint of the functor  $F_T$ . But notice that we have  $T(A) = A$  rather than merely  $T(A) \cong A$ . This is not so surprising as it may seem at first sight. Domain prelocales are only representations of

domains and what we are exploiting here is the simple idea that we can let  $A$  represent both  $D$  and  $F_T(D)$  via two *different* interpretation functions. Let us now address the question about canonicity. It suffices to check that the embedding corresponding to  $T(\mathbf{1}) \trianglelefteq T^2(\mathbf{1})$  is equal to  $F_T(e)$  where  $e: \mathbb{I} \rightarrow F_T(\mathbb{I})$  corresponds to  $\mathbf{1} \trianglelefteq T(\mathbf{1})$ . This is precisely the naturality of  $\tau$  which we listed as Step 15. It follows that the bilimit is the same as the one constructed in Chapter 5.

### 7.3.6 Languages for types, properties, and points

We define a formal language of *type* expressions by the following grammar:

$$\sigma ::= \mathbf{1} \mid X \mid (\sigma \rightarrow \sigma) \mid (\sigma \times \sigma) \mid (\sigma \oplus \sigma) \mid (\sigma)_{\perp} \mid \mathbf{P}^P(\sigma) \mid \text{rec} X.\sigma$$

where  $X$  ranges over a set  $TV$  of type variables. More constructions can be added to this list, of course, such as strict function space, smash product, Hoare powerdomain, and Smyth powerdomain. On the other hand, we do not include expressions for basic types, such as integers and booleans, as these can be encoded in our language by simple formulae.

We have seen two ways to interpret type expressions. The first interpretation takes values directly in  $\mathbf{B}$ , the category of bifinite domains, and is based on the constructions in Sections 3.2, 3.3, 5.1, and 6.2. Since a type expression may contain free variables, the interpretation can be defined only relative to an *environment*  $\rho_D: TV \rightarrow \mathbf{B}$ , which assigns to each type variable a bifinite domain. The semantic clauses corresponding to the individual rules of the grammar are as follows:

$$\begin{aligned} \mathcal{I}_D(\mathbf{1}; \rho_D) &= \mathbb{I}; \\ \mathcal{I}_D(X; \rho_D) &= \rho_D(X); \\ \mathcal{I}_D((\sigma \rightarrow \tau); \rho_D) &= [\mathcal{I}_D(\sigma; \rho_D) \longrightarrow \mathcal{I}_D(\tau; \rho_D)]; \\ &\text{etc.} \\ \mathcal{I}_D(\text{rec} X.\sigma; \rho_D) &= \text{FIX}(F_T), \\ &\text{where } F_T(E) = \mathcal{I}_D(\sigma; \rho_D[X \mapsto E]). \end{aligned}$$

The expression  $\rho_D[X \mapsto E]$  denotes the environment which maps  $X$  to  $E$  and coincides with  $\rho_D$  at all other variables.

Our work in the preceding subsections suggests that we can also interpret type expressions in the category **DomPreloc** of domain prelocales. Call the corresponding mappings  $\mathcal{I}_L$  and  $\rho_L$ . The semantic clauses for this localic interpretation are:

$$\begin{aligned} \mathcal{I}_L(\mathbf{1}; \rho_L) &= \mathbf{1}; \\ \mathcal{I}_L(X; \rho_L) &= \rho_L(X); \\ \mathcal{I}_L((\sigma \rightarrow \tau); \rho_L) &= [\mathcal{I}_L(\sigma; \rho_L) \rightarrow \mathcal{I}_L(\tau; \rho_L)]; \end{aligned}$$

etc.

$$\mathfrak{I}_L(\text{rec } X.\sigma; \rho_L) = \bigcup T^n(\mathbf{1}),$$

where  $T(A) = \mathfrak{I}_L(\sigma; \rho_L[X \mapsto A])$ .

The preceding subsections were meant to convince the reader of the following:

**Theorem 7.3.15.** *If  $\rho_L$  and  $\rho_D$  are environments such that for each  $X \in TV$  the domain prelocale  $\rho_L(X)$  is a localic description of  $\rho_D(X)$ , then for every type expression  $\sigma$  it holds that  $\mathfrak{I}_L(\sigma; \rho_L)$  is a localic description of  $\mathfrak{I}_D(\sigma; \rho_D)$ . As a formula:*

$$\text{spec}(\mathfrak{I}_L(\sigma; \rho_L)) \cong \mathfrak{I}_D(\sigma; \rho_D).$$

The next step is to define for each type expression  $\sigma$  a formal language  $\mathfrak{L}(\sigma)$  of (computational or observational) *properties*. This is done through the following inductive definition:

$$\begin{aligned} & \implies \text{true, false} \in \mathfrak{L}(\sigma); \\ \phi, \psi \in \mathfrak{L}(\sigma) & \implies \phi \wedge \psi, \phi \vee \psi \in \mathfrak{L}(\sigma); \\ \phi \in \mathfrak{L}(\sigma), \psi \in \mathfrak{L}(\tau) & \implies (\phi \rightarrow \psi) \in \mathfrak{L}(\sigma \rightarrow \tau), \\ \phi \in \mathfrak{L}(\sigma), \psi \in \mathfrak{L}(\tau) & \implies (\phi \times \psi) \in \mathfrak{L}(\sigma \times \tau); \\ \phi \in \mathfrak{L}(\sigma) & \implies (\phi \oplus \text{false}) \in \mathfrak{L}(\sigma \oplus \tau); \\ \psi \in \mathfrak{L}(\tau) & \implies (\text{false} \oplus \psi) \in \mathfrak{L}(\sigma \oplus \tau); \\ \phi \in \mathfrak{L}(\sigma) & \implies (\phi)_\perp \in \mathfrak{L}((\sigma)_\perp); \\ \phi \in \mathfrak{L}(\sigma) & \implies \Box\phi, \Diamond\phi \in \mathfrak{L}(\mathbf{P}^P(\sigma)); \\ \phi \in \mathfrak{L}(\sigma[\text{rec } X.\sigma/X]) & \implies \phi \in \mathfrak{L}(\sigma). \end{aligned}$$

Here we have used the expression  $\sigma[\tau/X]$  to denote the substitution of  $\tau$  for  $X$  in  $\sigma$ . The usual *caveat* about capture of free variables applies but let us not dwell on this. The rules exhibited above will generate for each  $\sigma$  the carrier set of a (syntactical) domain prelocale in the style of the previous subsections. Note that we don't need special properties for a recursively defined type as these are just the properties of the approximating domains bundled together (Theorem 7.3.14).

On each  $\mathfrak{L}(\sigma)$  we define a pre-order  $\lesssim$  and predicates  $\mathbf{C}$  and  $\mathbf{T}$  (the latter is needed for the coalesced sum construction) through yet another inductive definition. For example, the following axioms and rules enforce that each  $\mathfrak{L}(\sigma)$  is a pre-ordered distributive lattice.

$$\begin{aligned} & \implies \phi \lesssim \phi; \\ \phi \lesssim \psi, \psi \lesssim \chi & \implies \phi \lesssim \chi; \end{aligned}$$

$$\begin{aligned}
& \Rightarrow \phi \lesssim \text{true}; \\
\phi \lesssim \psi_1, \phi \lesssim \psi_2 & \Rightarrow \phi \lesssim \psi_1 \wedge \psi_2; \\
& \Rightarrow \phi \wedge \psi \lesssim \phi; \\
& \Rightarrow \phi \wedge \psi \lesssim \psi; \\
& \Rightarrow \text{false} \lesssim \phi; \\
\phi_1 \lesssim \psi, \phi_2 \lesssim \psi & \Rightarrow \phi_1 \vee \phi_2 \lesssim \psi; \\
& \Rightarrow \phi \lesssim \phi \vee \psi; \\
& \Rightarrow \psi \lesssim \phi \vee \psi; \\
& \Rightarrow \phi \wedge (\psi \vee \chi) \lesssim (\phi \wedge \psi) \vee (\phi \wedge \chi).
\end{aligned}$$

We have seen some type specific axioms and rules in the definition of the function space prelocale and the Plotkin powerlocale. For the full list we refer to [Abramsky, 1991b], p. 49ff. If  $\sigma$  is a closed type expression then the domain prelocale  $\mathcal{L}(\sigma)$  describes the intended bifinite domain:

**Theorem 7.3.16.** *If  $\sigma$  is a closed type expression then*

$$\text{spec}(\mathcal{L}(\sigma)) \cong \mathcal{I}_D(\sigma) .$$

(Note that this is a special case of Theorem 7.3.15.)

The whole scheme for deriving  $\lesssim$ ,  $\subset$ , and  $\top$  is designed carefully so as to have finite positive information in the premise of each rule only. Hence the whole system can be seen as a monotone inductive definition (in the technical sense of e.g. [Aczel, 1977]). Furthermore, we have already established close connections between the syntactical rules and properties of the described domains. This is the basis of the following result.

**Theorem 7.3.17.** *The language of properties is decidable.*

**Proof.** The statement is trivial for the domain prelocale  $\mathbf{1}$  because only combinations of true and false occur in  $\mathcal{L}(\mathbf{1})$ . For composite types we rely on the general development in Section 7.3.2, which at least for three concrete instances we have verified in Sections 7.3.3–5. First of all, every expression in  $\mathcal{L}(\sigma)$  can be effectively transformed into a finite disjunction of  $\vee$ -primes (i.e. expressions satisfying the C-predicate); this is Step 5, ‘prime generation’. Soundness and completeness ensure that the expressions satisfying the C-predicate are precisely the  $\vee$ -primes in the preordered lattice  $\mathcal{L}(\sigma)$ . Hence we can decide the preorder between arbitrary expressions if we can decide the preorder between  $\vee$ -primes. For the latter we note that our constructions accomplish more than we have stated so far. All  $\vee$ -primes, which are produced by the transformation algorithms, are of the explicit form occurring in the rules for deriving the C-predicate, rather than merely expressions which happen to be equivalent to  $\vee$ -primes. The pre-order between these explicit  $\vee$ -primes is (for each construction) easily characterized

through the semantic interpretation function  $\llbracket \cdot \rrbracket^0$ . The task of establishing the pre-order between these primes is then reduced to establishing some formula defined by structural induction on the type  $\sigma$ . Since every expression in  $\mathcal{L}(\sigma)$  is derived from true and false in finitely many steps, we will eventually have reduced our task to checking the preorder between certain expressions in  $\mathcal{L}(1)$ . ■

Finally, we introduce a formal language to speak about points of domains. So far, we have done this in a rather ad hoc way, trusting in the reader's experience with sets and functions. Doing it formally will allow us to establish a precise relationship between (expressions for) points and (expressions for) properties.

We assume that for each (closed) type expression  $\sigma$  we have a denumerable set  $V(\sigma) = \{x^\sigma, y^\sigma, z^\sigma, \dots\}$  of typed variables. The terms are defined as follows (where  $M : \sigma$  stands for ' $M$  is a term of type  $\sigma$ ')

$$\begin{aligned}
 & \Rightarrow *_\sigma : \sigma; \\
 & \Rightarrow x^\sigma : \sigma; \\
 & M : \tau \Rightarrow \lambda x^\sigma. M : (\sigma \rightarrow \tau); \\
 & M : (\sigma \rightarrow \tau), N : \sigma \Rightarrow (MN) : \tau; \\
 & M : \sigma, N : \tau \Rightarrow \langle M, N \rangle : (\sigma \times \tau); \\
 & M : (\sigma \times \tau), N : \nu \Rightarrow \text{let } M \text{ be } \langle x^\sigma, y^\tau \rangle. N : \nu; \\
 & M : \sigma \Rightarrow \text{inl}(M) : (\sigma \oplus \tau) \text{ and } \text{inr}(M) : (\tau \oplus \sigma); \\
 & M : (\sigma \oplus \tau), N_1 : \nu, N_2 : \nu \Rightarrow \text{cases } M \text{ of } \text{inl}(x^\sigma).N_1 \text{ else } \text{inr}(y^\tau).N_2 : \nu; \\
 & M : \sigma \Rightarrow \text{up}(M) : (\sigma)_\perp; \\
 & M : (\sigma)_\perp, N : \tau \Rightarrow \text{lift } M \text{ to } \text{up}(x^\sigma).N : \tau; \\
 & M : \sigma \Rightarrow \{M\} : P^P(\sigma); \\
 & M : P^P(\sigma), N : P^P(\tau) \Rightarrow \text{over } M \text{ extend } \{x^\sigma\}.N : P^P(\tau); \\
 & M : P^P(\sigma), N : P^P(\sigma) \Rightarrow M \sqcup N : P^P(\sigma); \\
 & M : P^P(\sigma), N : P^P(\tau) \Rightarrow M \otimes N : P^P(\sigma \times \tau); \\
 & M : \sigma[\text{rec } X. \sigma / X] \Rightarrow \text{fold}(M) : \text{rec } X. \sigma; \\
 & M : \text{rec } X. \sigma \Rightarrow \text{unfold}(M) : \sigma[\text{rec } X. \sigma / X]; \\
 & M : \sigma \Rightarrow \mu x^\sigma. M : \sigma.
 \end{aligned}$$

In the same fashion as for type expressions we have two alternatives for interpreting a term  $M$  of type  $\sigma$ . We can either give a direct denotational semantics in the bifinite domain  $\mathcal{I}_D(\sigma)$  or we can specify a prime filter in the corresponding domain prelocale  $\mathcal{L}(\sigma)$ . The denotational semantics suffers from the fact that in order to single out a particular element in a domain we use a mathematical language which looks embarrassingly similar to the



formal language we intend to interpret. Some of the semantic clauses to follow will therefore appear to be circular.

Again we need environments to deal with free variables. They are maps  $\rho: \bigcup_{\sigma} V(\sigma) \rightarrow \bigcup_{\sigma} \mathcal{I}_D(\sigma)$  which we assume to respect the typing. In the following clauses we will also suppress the type information.

$$\begin{aligned}
\llbracket *_{\sigma} \rrbracket \rho &= \perp, \text{ the least element in } \mathcal{I}_D(\sigma); \\
\llbracket x \rrbracket \rho &= \rho(x); \\
\llbracket \lambda x. M \rrbracket \rho &= (d \mapsto \llbracket M \rrbracket \rho[x \mapsto d]); \\
\llbracket (MN) \rrbracket \rho &= \llbracket M \rrbracket \rho(\llbracket N \rrbracket \rho); \\
\llbracket \langle M, N \rangle \rrbracket \rho &= \langle \llbracket M \rrbracket \rho, \llbracket N \rrbracket \rho \rangle; \\
\llbracket \text{let } M \text{ be } \langle x, y \rangle. N \rrbracket \rho &= \llbracket N \rrbracket \rho[x \mapsto d, y \mapsto e], \\
\text{where } d &= \pi_1(\llbracket M \rrbracket \rho), \\
e &= \pi_2(\llbracket M \rrbracket \rho); \\
\llbracket \text{inl}(M) \rrbracket \rho &= \text{inl}(\llbracket M \rrbracket \rho); \\
\llbracket \text{inr}(M) \rrbracket \rho &= \text{inr}(\llbracket M \rrbracket \rho); \\
\llbracket \text{cases } M \text{ of } \text{inl}(x).N_1 \text{ else } \text{inr}(y).N_2 \rrbracket \rho &= \begin{cases} \llbracket N_1 \rrbracket \rho[x \mapsto d], & \llbracket M \rrbracket \rho = (d: 1); \\ \llbracket N_2 \rrbracket \rho[y \mapsto e], & \llbracket M \rrbracket \rho = (e: 2); \\ \perp, & \llbracket M \rrbracket \rho = \perp; \end{cases} \\
\llbracket \text{up}(M) \rrbracket \rho &= \text{up}(\llbracket M \rrbracket \rho); \\
\llbracket \text{lift } M \text{ to } \text{up}(x^{\sigma}).N \rrbracket \rho &= \begin{cases} \llbracket N \rrbracket \rho[x \mapsto d], & \llbracket M \rrbracket \rho = \text{up}(d); \\ \perp, & \llbracket M \rrbracket \rho = \perp; \end{cases} \\
\llbracket \{M\} \rrbracket \rho &= \{\llbracket M \rrbracket \rho\}; \\
\llbracket \text{over } M \text{ extend } \{x^{\sigma}\}.N \rrbracket \rho &= \uparrow X \cap \text{Cl}(X), \\
\text{where } X &= \bigcup \{\llbracket N \rrbracket \rho[x \mapsto d] \mid d \in \llbracket M \rrbracket \rho\}; \\
\llbracket M \sqcup N \rrbracket \rho &= \llbracket M \rrbracket \rho \sqcup \llbracket N \rrbracket \rho; \\
\llbracket M \otimes N \rrbracket \rho &= \{\langle d, e \rangle \mid d \in \llbracket M \rrbracket \rho, e \in \llbracket N \rrbracket \rho\}; \\
\llbracket \text{fold}(M) \rrbracket \rho &= \text{fold}(\llbracket M \rrbracket \rho); \\
\llbracket \text{unfold}(M) \rrbracket \rho &= \text{unfold}(\llbracket M \rrbracket \rho); \\
\llbracket \mu x. M \rrbracket \rho &= \text{fix}(f), \\
\text{where } f(d) &= \llbracket M \rrbracket \rho[x \mapsto d].
\end{aligned}$$

Now let us give the localic, or, as we are now justified in saying, *logical* interpretation. We use a sequent calculus style of presenting this *domain logic*. The problem of free variables is dealt with this time by including a finite list  $\Gamma$  of assumptions on variables. We write them in the form  $x \mapsto \phi$  and assume that  $\Gamma$  contains at most one of these for each variable  $x$ . A sequent then takes the form  $\Gamma \vdash M : \phi$  and should be read as ‘ $M$  satisfies  $\phi$

under the assumptions in  $\Gamma'$ .

$$\begin{array}{ll}
\{\Gamma \vdash M : \phi_i\}_{i \in I} \implies \Gamma \vdash M : \bigwedge_{i \in I} \phi; \\
\phi' \lesssim \phi, \psi \lesssim \psi', \\
(\Gamma, x \mapsto \phi \vdash M : \psi) \implies \Gamma, x \mapsto \phi' \vdash M : \psi'; \\
\{\Gamma, x \mapsto \phi_i \vdash M : \psi\}_{i \in I} \implies \Gamma, x \mapsto \bigvee_{i \in I} \phi_i \vdash M : \psi; \\
\Gamma \vdash M : \psi \implies \Gamma, x \mapsto \phi \vdash M : \psi; \\
\implies x \mapsto \phi \vdash x : \phi; \\
\Gamma, x \mapsto \phi \vdash M : \psi \implies \Gamma \vdash \lambda x. M : (\phi \rightarrow \psi); \\
\Gamma \vdash M : (\phi \rightarrow \psi); \Gamma \vdash N : \phi \implies \Gamma \vdash (MN) : \psi; \\
\Gamma \vdash M : \phi; \Gamma \vdash N : \psi \implies \Gamma \vdash \langle M, N \rangle : (\phi \times \psi); \\
\Gamma \vdash M : (\phi \times \psi), \\
\Gamma, x \mapsto \phi, y \mapsto \psi \vdash N : \chi \implies \Gamma \vdash \text{let } M \text{ be } \langle x, y \rangle. N : \chi; \\
\Gamma \vdash M : \phi \implies \Gamma \vdash \text{inl}(M) : (\phi \oplus \text{false}); \\
\Gamma \vdash M : \phi \implies \Gamma \vdash \text{inr}(M) : (\text{false} \oplus \phi); \\
\Gamma \vdash M : (\phi \oplus \text{false}), \top(\phi), \\
\Gamma, x \mapsto \phi \vdash N_1 : \psi \implies \Gamma \vdash \text{cases } M \text{ of } \text{inl}(x).N_1 \\
\text{else } \text{inr}(y).N_2 : \psi; \\
\Gamma \vdash M : (\text{false} \oplus \phi), \top(\phi), \\
\Gamma, y \mapsto \phi \vdash N_2 : \psi \implies \Gamma \vdash \text{cases } M \text{ of } \text{inl}(x).N_1 \\
\text{else } \text{inr}(y).N_2 : \psi; \\
\Gamma \vdash M : \phi \implies \Gamma \vdash \text{up}(M) : (\phi)_{\perp}; \\
\Gamma \vdash M : (\phi)_{\perp}; \Gamma, x \mapsto \phi \vdash N : \psi \implies \Gamma \vdash \text{lift } M \text{ to } \text{up}(x^{\sigma}).N : \psi; \\
\Gamma \vdash M : \phi \implies \Gamma \vdash \llbracket M \rrbracket : \Box \phi; \\
\Gamma \vdash M : \phi \implies \Gamma \vdash \llbracket M \rrbracket : \Diamond \phi; \\
\Gamma \vdash M : \Box \phi; \Gamma, x \mapsto \phi \vdash N : \Box \psi \implies \Gamma \vdash \text{over } M \text{ extend } \llbracket x^{\sigma} \rrbracket. N : \Box \psi; \\
\Gamma \vdash M : \Diamond \phi; \Gamma, x \mapsto \phi \vdash N : \Diamond \psi \implies \Gamma \vdash \text{over } M \text{ extend } \llbracket x^{\sigma} \rrbracket. N : \Diamond \psi; \\
\Gamma \vdash M : \Box \phi; \Gamma \vdash N : \Box \phi \implies \Gamma \vdash M \sqcup N : \Box \phi; \\
\Gamma \vdash M : \Diamond \phi \implies \Gamma \vdash M \sqcup N : \Diamond \phi; \\
\Gamma \vdash N : \Diamond \phi \implies \Gamma \vdash M \sqcup N : \Diamond \phi; \\
\Gamma \vdash M : \Diamond \phi; \Gamma \vdash N : \Diamond \psi \implies \Gamma \vdash M \otimes N : \Diamond(\phi \times \psi); \\
\Gamma \vdash M : \Box \phi; \Gamma \vdash N : \Box \psi \implies \Gamma \vdash M \otimes N : \Box(\phi \times \psi); \\
\Gamma \vdash M : \phi \implies \Gamma \vdash \text{fold}(M) : \phi; \\
\Gamma \vdash M : \phi \implies \Gamma \vdash \text{unfold}(M) : \phi;
\end{array}$$

$$\Gamma \vdash \mu x.M : \phi; \Gamma, x \mapsto \phi \vdash M : \psi \implies \Gamma \vdash \mu x.M : \psi.$$

A few comments may help in reading these clauses. The first two rules guarantee that the set of properties which can be deduced for a term  $M$  forms a filter in the domain prelocale. The third rule expresses the fact that every particular  $x$  will satisfy properties from a prime filter. In particular, it entails that  $\Gamma, x \mapsto \text{false} \vdash M : \phi$  is always true. The fourth rule (which is the last of the structural rules) is ordinary weakening. We need it to get started in a derivation. In the two rules for the cases-construct the predicate  $\top$  shows up. Instead of  $\top(\phi)$  we could have written  $\phi \not\approx \text{false}$  but as we said before, we want to keep the whole logic positive, that is to say, we want to use inductive definitions only. The two rules for fold and unfold may seem a bit boring, but it is precisely at this point where we take advantage of the fact that in the world of domain prelocales we solve the domain equation up to equality. The last rule, finally, has to be applied finitely many times, starting from  $\Gamma \vdash \mu x.M : \text{true}$ , in order to yield something interesting. Here we may note with regret that our whole system is based on the logic of observable properties. A standard proof principle such as fixpoint induction for admissible predicates, Lemma 2.1.20, does not fit into the framework. On the other hand, it is hopefully apparent how canonical the whole approach is. For applications, see [Abramsky, 1990c; Abramsky, 1991a; Boudol, 1991; Hennessy, 1993; Ong, 1993; Jensen, 1991; Jensen, 1992].

Let us now compare denotational and logical semantics. We need to say how environments  $\rho$  and assumptions  $\Gamma$  fit together. First of all, we assume that  $\rho$  maps each variable  $x^\sigma$  into  $\text{spec}(\mathcal{L}(\sigma))$ . Secondly, we want that  $\rho(x)$  belongs to the compact-open subset described by the corresponding entry in  $\Gamma$ . But since environments are functions defined on the whole set of variables while assumptions are finite lists, the following definition is a bit delicate. We write  $\rho \models \Gamma$  if for all entries  $x \mapsto \phi$  in  $\Gamma$  we have  $\rho(x) \in \llbracket \phi \rrbracket$ . Using this convention, we can formulate validity for assertions about terms:

$$\Gamma \models M : \phi \text{ if and only if } \forall \rho. (\rho \models \Gamma \implies \llbracket M \rrbracket \rho \in \llbracket \phi \rrbracket) .$$

The final tie-up between the two interpretations of type expressions and terms then is the following:

**Theorem 7.3.18.** *The domain logic is sound and complete. As a formula:*

$$\forall M, \Gamma, \phi. \quad \Gamma \vdash M : \phi \quad \text{if and only if} \quad \Gamma \models M : \phi .$$

### Exercises 7.3.19.

1. Prove that a completely distributive lattice also satisfies the dual distributivity axiom:  $\bigvee_{i \in I} \bigwedge A_i = \bigwedge_{f: I \rightarrow \bigcup A_i} \bigvee_{i \in I} f(i)$ .

2. [Raney, 1960] Prove that a complete lattice  $L$  is completely distributive if and only if the following holds for all  $x \in L$ :

$$x = \bigvee_{a \not\leq x} \bigwedge_{b \not\leq a} b.$$

(Hint: Use Theorem 7.1.3.)

3. Show that a topological space is sober if and only if every irreducible closed set is the closure of a unique point.
4. Find a complete lattice  $L$  for which  $\text{pt}(L)$  is empty.
5. Show that every Hausdorff space is sober. Find a  $T_1$ -space which is not sober. The converse, a sober space, which is not  $T_1$ , ought to be easy to find.
6. Find a deppo which is not sober in the Scott-topology. (Reference: [Johnstone, 1981]. For an example which is a complete lattice, see [Isbell, 1982]. There is no known example which is a distributive lattice.)
7. Describe the topological space  $\text{pt}(L)$  in terms of  $\wedge$ -prime elements of the complete lattice  $L$ .
8. Let  $D$  be a continuous domain. Identify  $D$  with the set of  $\wedge$ -prime elements in  $\Omega(D)$ . Prove that the Lawson-topology on  $D$  is the restriction of the Lawson-topology on  $\Omega(D)$  to  $D$ .
9. Suppose  $f: V \rightarrow W$  is a lattice homomorphism. Show that  $R$  defined by  $xRy$  if  $y \leq f(x)$  is a join-approximable relation. Characterize the continuous functions between spectral spaces which arise from these particular join-approximable relations.
10. Extend Lemma 7.3.8 to other classes of domains.
11. Try to give a localic description of the coalesced sum construction.

## 8 Further directions

Our coverage of domain theory is by no means comprehensive. Twenty-five years after its inception, the field remains extremely active and vital. We shall try in this section to give a map of the parts of the subject we have not covered.

### 8.1 Further topics in ‘classical domain theory’

We mention four topics which the reader is likely to encounter elsewhere in the literature.

#### 8.1.1 Effectively given domains

As we mentioned in the Introduction, domain-theoretic continuity provides a qualitative substitute for explicit computability considerations. In order

to evaluate this claim rigorously, one should give an effective version of domain theory, and check that the key constructions on domains such as product, function space, least fixpoints, and solutions of recursive domain equations, all ‘lift’ to this effective setting. For this purpose, the use of abstract bases becomes quite crucial; we say (simplifying a little for this thumbnail sketch) that an  $\omega$ -continuous domain is *effectively given* if it has an abstract basis  $(B, \prec)$  which is numbered as  $B = \{b_n\}_{n \in \omega}$  in such a way that  $\prec$  is recursive in the indices. Similarly, a continuous function  $f: D \rightarrow E$  between effectively given domains is effective if the corresponding approximable mapping is recursively enumerable. We refer to [Smyth, 1977; Kanda, 1979; Weihrauch and Deil, 1980] and the chapter on Effective Structures in this Handbook for developments of effective domain theory on these lines.

There have also been some more sophisticated approaches which aim at making effectivity ‘intrinsic’ by working inside a constructive universe for set theory based on recursive realizability [McCarty, 1984; Rosolini, 1986; Phoa, 1991]. We shall return to this idea in Section 8.5.

### 8.1.2 Universal Domains

Let  $\mathbf{C}$  be a cartesian closed category of domains, and  $U$  a domain in  $\mathbf{C}$ . We say that  $U$  is *universal* for  $\mathbf{C}$  if, for every  $D$  in  $\mathbf{C}$ , there is an embedding  $e: D \rightarrow U$ . Thus universality means that we can, in effect, replace the category  $\mathbf{C}$  by the single domain  $U$ . More precisely, we can regard the domain  $D$  as represented by the idempotent  $e_D = e \circ p$ , where  $p$  is the projection corresponding to  $e$ . Since  $e_D: U \rightarrow U$ , and  $[U \rightarrow U]$  is again in  $\mathbf{C}$  and hence embeddable in  $U$ , we can ultimately identify  $D$  with an *element*  $u_D \in U$ , which we can think of as a ‘code’ for  $D$ . Moreover, constructions such as product and function space induce continuous functions

$$\text{fun, prod} : U^2 \rightarrow U$$

which act on these codes, so that, for example,

$$\text{fun}(u_D, u_E) = u_{[D \rightarrow E]}.$$

In this way, the whole functorial level of domain theory which we developed as a basis for the solution of recursive domain equations in Section 5 can be eliminated, and we can solve domain equations *up to equality on the codes* by finding fixpoints of continuous functions over  $U$ .

This approach was introduced by Scott in [Scott, 1976], and followed in the first textbook on denotational semantics [Stoy, 1977]. However, it must be said that, as regards applications, universal domains have almost fallen into disuse. The main reason is probably that the coding involved in the transition from  $D$  to  $u_D$  is confusing and unappealing; while more



attractive ways of simplifying the treatment of domain equations, based on information systems, have been found (see Section 8.1.4). However, there have been two recent developments of interest. Firstly, a general approach to the construction of universal domains, using tools from Model Theory, has been developed by Gunter and Jung and Droste and Göbel, and used to construct universal domains for many categories, and to prove their non-existence in some cases [Gunter and Jung, 1988] and Droste and Göbel [1990; 1991; 1993].

Secondly, there is one application where universal domains do play an important rôle: to provide models for type theories with a type of all types. Again, the original idea goes back to [Scott, 1976]. We say that a universal domain  $U$  admits a universal type if the subdomain  $V$  of all  $u_D$  for  $D$  in  $\mathbf{C}$  is itself a domain in  $\mathbf{C}$ —and hence admits a representation  $u_V \in U$ . We can think of  $u_V$  as a code for the type of all types. In [Scott, 1976], Scott studied the powerset  $\mathfrak{P}(\omega)$  as a universal domain for two categories: the category of  $\omega$ -continuous lattices (for which domains are taken to be represented by idempotents on  $\mathfrak{P}(\omega)$ ), and the category of  $\omega$ -algebraic lattices (for which domains are represented by closures). Ershov [1975] and Hosono and Sato [1977] independently proved that  $\mathfrak{P}(\omega)$  does not admit a universe for the former category; Hancock and Martin-Löf proved that it does for the latter (reported in [Scott, 1976]). For recent examples of the use of universal domains to model a type of all types see [Taylor, 1987; Coquand, 1989; Berardi, 1991].

### 8.1.3 Domain-theoretic semantics of polymorphism

We have seen the use of continuity in domain theory to circumvent cardinality problems in finding solutions to domain equations such as

$$D \cong [D \longrightarrow D] .$$

A much more recent development makes equally impressive use of continuity to give a finitary semantics for impredicative polymorphism, as in the second-order *lambda*-calculus (Girard's 'System F') [Girard, 1986; Coquand *et al.*, 1987; Coquand, 1989]. This semantics makes essential use of the functorial aspects of Domain Theory. There have also been semantics for implicit polymorphism based on ideals [MacQueen *et al.*, 1986] and partial equivalence relations [Abadi and Plotkin, 1990] over domains. We refer to the chapter in this volume of the Handbook on Semantics of Types for comprehensive coverage and references.

#### 8.1.4 Information systems

Scott introduced information systems for bounded-complete  $\omega$ -algebraic dcpos ('Scott domains') in [Scott, 1982]. The idea is, roughly, to represent a category of domains by a category of abstract bases and approximable

mappings as in Theorems 2.2.28 and 2.2.29. One can then define constructions on domains in terms of the bases, as in Propositions 3.2.4 and 4.2.4. This gives a natural setting for effective domain theory as in Section 8.1.1. Moreover, bilimits are given by unions of information systems, and domain equations solved up to equality, much as in Section 7.3.5. More generally, information systems correspond to presenting just the coprime elements from the domain prelocales of Section 7.3. Information system representations of various categories of domains can be found in [Winskel, 1988; Zhang, 1991; Curien, 1993]. A general theory of information systems applicable to a wide class of topological and metric structures can be found in [Edalat and Smyth, 1993].

## 8.2 Stability and sequentiality

Recall the  $\epsilon$ - $\delta$  style definition of continuity given in Proposition 2.2.11: given  $e \in C_{f(x)}$  it provides  $d \in B_x$  with  $f(d) \sqsubseteq e$ . However, there is no *canonical* choice of  $d$  from  $e$ . In an order-theoretic setting, it is natural to ask for there to be a *least* such  $d$ . This leads to the idea of the *modulus of stability*:  $M(f, x, e)$ , where  $f(x) \sqsupseteq e$ , is the least such  $d$ , if it exists. We say that a continuous function is *stable* if the modulus always exists, and define the *stable ordering* on such functions by

$$f \sqsubseteq_s g \iff f \sqsubseteq g \wedge \forall x, e. e \in C_{f(x)}. M(f, x, e) = M(g, x, e).$$

We can think of the modulus as specifying the minimum information actually required of a given input  $x$  in order that the function  $f$  yields a given information  $y$  on the output; the stable ordering refines the usual pointwise order by taking this intensional information into account.

It turns out that these definitions are equivalent to elegant algebraic notions in the setting of the lattice-like domains introduced (for completely different purposes!) in Section 4.1. Let  $D, E$  be domains in  $\mathbf{L}$ . Then a continuous function  $f: D \rightarrow E$  is stable iff it preserves bounded non-empty infima (which always exist in  $\mathbf{L}$ ; cf. Proposition 4.1.2), and  $f \sqsubseteq_s g$  iff for all  $x \sqsubseteq y$ ,  $f(x) = f(y) \sqcap g(x)$ . This is the first step in an extensive development of 'stable domain theory' in which stable functions under the stable ordering take the place which continuous functions play in standard domain theory. Stable domain theory was introduced by Berry [1978; 1979]. Some more recent references are [Girard, 1986; Coquand *et al.*, 1987; Taylor, 1990; Ehrhard, 1993].

Berry's motivation in introducing stable functions was actually to try to capture the notion of sequentially computable function at higher types. For the theory of sequential functions on concrete domains, we refer to [Kahn and Plotkin, 1993; Curien, 1993].

### 8.3 Reformulations of domain theory

At various points in our development of domain theory (see e.g. Section 3.2), we have referred to the need to switch between different versions  $\mathbf{C}$ ,  $\mathbf{C}_\perp$ ,  $\mathbf{C}_{\perp!}$  of some category of domains, depending on whether bottom elements are required, and if so whether functions are required to preserve them. In some sense  $\mathbf{C}$  and  $\mathbf{C}_{\perp!}$  are the mathematically natural categories, since what the morphisms must preserve matches the structure that the objects are required to have; while  $\mathbf{C}_\perp$  is the preferred category for semantics, since endomorphisms  $f: D \rightarrow D$  need not have fixpoints at all in  $\mathbf{C}$ , while least fixpoints in  $\mathbf{C}_{\perp!}$  are necessarily trivial.

All this suggests that something is lacking from the mathematical framework in order to get a really satisfactory tie-up with the applications. We shall describe a number of attempts to make good this deficiency. While no definitive solution has yet emerged, these proposals have contributed important insights to domain theory and its applications.

#### 8.3.1 Predomains and partial functions

The first proposal is due to Gordon Plotkin [1985]. The idea is to use the *objects* of  $\mathbf{C}$  ('predomains', i.e. domains without any requirement of bottom elements), but to change the notion of morphism to *partial continuous function*, where we say that a partial function  $f: D \rightarrow E$  is continuous if its domain of definition is a Scott-open subset of  $D$ , and its restriction to this subset is a (total) continuous function. The resulting category is denoted by  $\mathbf{C}_\partial$ . This switch to partial continuous functions carries with it a change in the type structure we can expect to have in our categories of domains: they should be *partial* cartesian closed categories, as defined e.g. in [Robinson and Rosolini, 1988; Rosolini, 1986].

One advantage of this approach is that it brings the usage of domain theory closer to that of recursion theory. For example, the hierarchy of (strict) partial continuous functionals over the natural numbers will be given by

$$\mathbf{N}, [\mathbf{N} \rightarrow \mathbf{N}], [[\mathbf{N} \rightarrow \mathbf{N}] \rightarrow \mathbf{N}], \dots$$

rather than

$$\mathbf{N}_\perp, [\mathbf{N}_\perp \xrightarrow{\perp!} \mathbf{N}_\perp], [[\mathbf{N}_\perp \xrightarrow{\perp!} \mathbf{N}_\perp] \xrightarrow{\perp!} \mathbf{N}_\perp], \dots$$

This avoidance of bottom elements also leads to a simpler presentation of product and sum types. For example, there is just one notion of sum, the disjoint union  $D \dot{\cup} E$ , which is indeed the coproduct in  $\mathbf{C}_\partial$ .

An important point is that there is a good correspondence between the operational behaviour of functions with a call-by-value parameter-passing mechanism and the partial function type  $[\_ \rightarrow \_]$ . For example, there is a good fit between  $[\_ \rightarrow \_]$  and the function type constructor in Standard ML [Milner and Tofte, 1991; Milner *et al.*, 1990].

To balance these advantages, we have the complication of dealing with partially defined expressions and partial cartesian closure; and also a less straightforward treatment of fixpoints. It is not the case that an arbitrary partial continuous function  $f: D \rightarrow D$  has a well-defined least fixpoint. However, if  $D$  itself is a partial function type, e.g.  $D = [E \rightarrow E]$ , then  $f$  does have a well-defined least fixpoint. This is in accord with computational intuition for call-by-value programming languages, but not so pleasant mathematically.

As a final remark, note that in fact  $\mathbf{C}_\partial$  is equivalent to  $\mathbf{C}_{\perp!}$ ! Thus, in a sense, this approach brings nothing new. However, there is a distinct conceptual difference, and also  $\mathbf{C}_\partial$  is more amenable to constructive proof and categorical axiomatization [Rosolini, 1986].

### 8.3.2 Computational monads

Computational monads have been proposed by Eugenio Moggi as a general structuring mechanism for denotational semantics [Moggi, 1991]. A computational monad on a cartesian category  $\mathbf{C}$  is a monad  $(T, \eta, \mu)$  together with a ‘tensorial strength’, i.e. a natural transformation

$$t_{A,B} : A \times TB \rightarrow T(A \times B)$$

satisfying some equational axioms. The import of the strength is that the monad can be internalized along the lines mentioned after Proposition 6.1.8. Now let  $\mathbf{C}$  be a category of (pre)domains and total continuous functions. Moggi’s proposal is to make a distinction between *values* and (denotations of) *computations*. An element of  $A$  is a value, an element of  $TA$  is a computation. A (call-by-value) procedure will denote a morphism  $A \rightarrow TB$  which accepts an input value of type  $A$  and produces a computation over  $B$ . Composition of such morphisms is by Kleisli extension: if  $f: A \rightarrow TB$ ,  $g: B \rightarrow TC$ , then composition is defined by

$$A \xrightarrow{f} TB \xrightarrow{Tg} TTC \xrightarrow{\mu_C} TC,$$

with identities given by the unit  $\eta_A : A \rightarrow TA$ .

In particular, partiality can be captured in this way using the *lifting* monad, for which see Section 3.2.5. Note that this particular example is really just another way of presenting the category  $\mathbf{C}_\partial$  of the previous subsection; there is a natural isomorphism

$$[D \rightarrow E_\perp] \cong [D \rightarrow E].$$

The value of the monadic approach lies in its generality and in the type distinction it introduces between values and computations. To illustrate the first point, note that the various powerdomain constructions presented



in Section 7.2 all have a natural structure as strong monads, with the monad unit and multiplication given by suitable versions of the singleton and big union operations. For the second point, we refer to the elegant axiomatization of general recursion in terms of fixpoint objects given by Crole and Pitts [1992], which makes strong use of the monadic approach. This work really belongs to axiomatic domain theory, to which we will return in Section 8.4 below.

### 8.3.3 Linear Types

Another proposal by Gordon Plotkin is to use linear types (in the sense of linear logic [Girard, 1987]) as a metalanguage for domain theory [Plotkin, 1993]. This is based on the following observation. Consider a category  $\mathbf{C}_{\perp!}$  of domains with bottom elements and strict continuous functions. This category has products and coproducts, given by cartesian products and coalesced sums. It also has a monoidal closed structure given by smash product and strict function space, as mentioned in Section 3.2.4. Now lifting, which is a monad on  $\mathbf{C}$  by virtue of the adjunction mentioned in Section 3.2.5, is dually a *comonad* on  $\mathbf{C}_{\perp!}$ ; and the co-Kelisi category for this comonad is  $\mathbf{C}_{\perp}$ .

Indeed, linear logic has broader connections with domain theory. A key idea of linear logic is the linear decomposition of the function space:

$$[A \longrightarrow B] \cong [!A \multimap B] .$$

One of the cardinal principles of domain theory, as we have seen, is to look for cartesian closed categories of domains as convenient universes for the semantics of computation. Linear logic leads us to look for linear decompositions of these cartesian closed structures. For example, the cartesian closed category of complete lattices and continuous maps has a linear decomposition via the category of complete lattices and sup-lattice homomorphisms—i.e. maps preserving *all* joins, with  $!L = P^H(L)$ , the Hoare powerdomain of  $L$ . There are many other examples [Hoofman, 1992; Ehrhard, 1993; Huth, 1994].

## 8.4 Axiomatic domain theory

We began our account of domain theory with requirements to interpret certain forms of recursive definitions, and to abstract some key structural features of computable partial functions. We then introduced some quite specific structures for convergence and approximation. The elaboration of the resulting theory showed that these structures do indeed *work*; they meet the requirements with which we began. The question remains whether another class of structures might have served us well or better. To address this question, we should try to axiomatize the key features of a category of domains which make it suitable to serve as a universe for the semantics



of computation. Such an exercise may be expected to yield the following benefits:

- By making it clearer what the essential structure is, it should lead to an improved meta-language and logic, a refinement of Scott's logic of computable functions [Scott, 1993].
- Having a clear axiomatization might lead to the discovery of different models, which might perhaps be more convenient for certain purposes, or suggest new applications. On the other hand, it might lead to a representation theorem, to the effect that every model of our axioms for a 'category of domains' can in fact be embedded in one of the concrete categories we have been studying in this chapter.

Thus far, only a limited amount of progress has been made on this programme. One step that can be made relatively cheaply is to generalize from concrete categories of domains to categories enriched over some suitable subcategory of **DCPO**. Much of the force of domain theory carries over directly to this more general setting [Smyth and Plotkin, 1982; Freyd, 1992]. Moreover, this additional generality is not spurious. A recent development in the semantics of computation has been towards a refinement of the traditional denotational paradigm, to reflect more intensional aspects of computational behaviour. This has led to considering as semantic universes certain categories in which the morphisms are not functions but sequential algorithms [Curien, 1993], information flows [Abramsky and Jagadeesan, 1994b], game-theoretic strategies [Abramsky and Jagadeesan, 1994a], or concurrent processes [Abramsky, 1994]. These are quite different from the 'concrete' categories of domains we have been considering, in which the morphisms are always functions. Nevertheless, they have many of the relevant properties of categories of domains, notably the existence of fixpoints and of canonical solutions of recursive domain equations. The promise of axiomatic domain theory is to allow the rich theory we have developed in this chapter to be transposed to such settings with a minimum of effort.

The most impressive step towards axiomatic domain theory to date has been Peter Freyd's work on algebraically compact categories [1991; 1992]. This goes considerably beyond what we covered in Section 5. The work by Crole and Pitts on FIX-categories should also be mentioned [1992].

In another direction, there are limitative results which show that certain kinds of structures *cannot* serve as categories of domains. One such result appeared as Exercise 5.4.11(3). For another, see Hofmann and Mislove [1993].

## 8.5 Synthetic domain theory

A more radical conceptual step is to try to absorb all the structure of convergence and approximation, indeed of computability itself, into the am-

bient universe of sets, by working inside a suitable constructive set theory or topos. The slogan is: ‘domains are sets’. This leads to a programme of ‘synthetic domain theory’, by analogy with synthetic differential geometry [Kock, 1981], in which smoothness rather than effectivity is the structure absorbed into the ambient topos.

The programme of synthetic domain theory was first adumbrated by Dana Scott around 1980. First substantial steps on this programme were taken by Rosolini [1986], and subsequently by Phoa [1991], and Freyd, Mulry, Rosolini and Scott [Freyd *et al.*, 1990]. Axioms for synthetic domain theory have been investigated by Hyland [1991] and Taylor [1991], and the subject is currently under active development.

## 9 Guide to the literature

As mentioned in the Introduction, there is no book on domain theory. For systematic accounts by the two leading contributors to the subject, we refer to the lecture notes of Scott [1981] and Plotkin [Plotkin, 1981]. There is also an introductory exposition by Gunter and Scott in [1990]. An exhaustive account of the theory of continuous lattices can be found in [Gierz *et al.*, 1980]; a superb account of Stone duality, with a good chapter on continuous lattices, is given in [Johnstone, 1982]; while Davey and Priestly [1990] is an excellent and quite gentle introduction to the theory of partial orders.

Some further reading on the material covered in this Chapter:

**Section 2:** [Davey and Priestley, 1990; Johnstone, 1982];

**Section 3:** [Plotkin, 1981; Gunter, 1992b; Winskel, 1993];

**Section 4:** [Jung, 1989; Jung, 1990];

**Section 5:** [Smyth and Plotkin, 1982; Freyd, 1991; Freyd, 1992; Pitts, 1993a; Pitts, 1993b];

**Section 6:** [Plotkin, 1976; Smyth, 1978; Winskel, 1983; Heckmann, 1991; Schalk, 1993];

**Section 7:** [Abramsky, 1990c; Abramsky, 1991a; Abramsky and Ong, 1993; Ong, 1993; Hennessy, 1993; Boudol, 1991; Jensen, 1992; Jensen, 1991; Smyth, 1983b].

## Applications of domain theory

There is by now an enormous literature on the semantics of programming languages, much of it using substantial amounts of Domain Theory. We will simply list a number of useful textbooks: [Schmidt, 1986; Tennent, 1991; Gunter, 1992b; Winskel, 1993].

In addition, a number of other applications of domain theory have arisen: in Abstract Interpretation and static program analysis [Abramsky, 1990a; Burn *et al.*, 1986; Abramsky and Jensen, 1991] (see also the article on Abstract Interpretation in this Handbook); databases Buneman *et al.*

[1988; 1991]; computational linguistics [Pereira and Shieber, 1984; Pollard and Moshier, 1990]; artificial intelligence [Rounds and Zhang, 1994]; fractal image generation [Edalat, 1993b]; and foundations of analysis [Edalat, 1993a].

Finally, the central importance of domain theory is well indicated by the number of other chapters of this Handbook which make substantial reference to Domain-theoretic ideas: topology, algebraic semantics, semantics of types, correspondence between operational and denotational semantics, abstract interpretation, effective structures.

## References

- [Abadi and Plotkin, 1990] M. Abadi and G. D. Plotkin. A Per model of polymorphism and recursive types. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 355–365. IEEE Computer Society Press, 1990.
- [Abramsky, 1987] S. Abramsky. *Domain Theory and the Logic of Observable Properties*. PhD thesis, University of London, 1987.
- [Abramsky, 1990a] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–40, 1990.
- [Abramsky, 1990b] S. Abramsky. A generalized Kahn principle for abstract asynchronous networks. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 1990.
- [Abramsky, 1990c] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.
- [Abramsky, 1991a] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92:161–218, 1991.
- [Abramsky, 1991b] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.
- [Abramsky, 1994] S. Abramsky. Interaction categories and communicating sequential processes. In A. W. Roscoe, editor, *A Classical Mind: essays in honour of C. A. R. Hoare*, chapter 1, pages 1–16. Prentice-Hall International, 1994.
- [Abramsky and Jagadeesan, 1994a] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 1994. To appear.
- [Abramsky and Jagadeesan, 1994b] S. Abramsky and R. Jagadeesan. New foundations for the geometry of interaction. *Information and Computation*, 1994. To appear.

- [Abramsky and Jensen, 1991] S. Abramsky and T. Jensen. A relational approach to strictness analysis for polymorphic functions. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 49–54. ACM Press, 1991.
- [Abramsky and Ong, 1993] S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.
- [Aczel, 1977] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *The Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and Foundations of Mathematics*, pages 739–782. North-Holland, 1977.
- [Adámek and Trnková, 1989] J. Adámek and V. Trnková. *Automata and Algebras in Categories*. Kluwer Academic Publishers, 1989.
- [Adámek et al., 1982] J. Adámek, E. Nelson, and J. Reiterman. Tree constructions of free continuous algebras. *Journal of Computer and System Sciences*, 24:114–146, 1982.
- [Arden, 1960] D. N. Arden. *Theory of Computing Machine Design*, chapter Delayed logic and finite state machines. University of Michigan Press, 1960.
- [Barendregt, 1984] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [Bekič, 1969] H. Bekič. Definable operations in general algebras, and the theory of automata and flowcharts. Technical report, IBM Laboratory, Vienna, 1969. Reprinted in H. Bekič, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer Verlag, 1984.
- [Bekič, 1971] H. Bekič. Towards a mathematical theory of programs. Technical Report TR 25.125, IBM Laboratory, Vienna, 1971.
- [Berardi, 1991] S. Berardi. Retractions on dI-domains as a model for type:type. *Information and Computation*, 94:204–231, 1991.
- [Berry, 1978] G. Berry. Stable models of typed  $\lambda$ -calculi. In *Proceedings of the 5th International Colloquium on Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, pages 72–89. Springer-Verlag, 1978.
- [Berry, 1979] G. Berry. Modèles Complément Adéquats et Stables des Lambda-calculs typés, 1979. Thèse de Doctorat d'Etat, Université Paris VII.
- [Boudol, 1991] G. Boudol. Lambda calculi for (strict) parallel functions. Technical Report 1387, INRIA - Sophia Antipolis, 1991. To appear in *Information and Computation*.
- [Buneman et al., 1988] P. Buneman, S. Davidson, and A. Watters. A semantics for complex objects and approximate queries. In *Principles of*



*Database Systems*. ACM, 1988.

- [Buneman *et al.*, 1991] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 91:23–55, 1991.
- [Burn *et al.*, 1986] G. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.
- [Coquand, 1989] T. Coquand. Categories of embeddings. *Theoretical Computer Science*, 68:221–238, 1989.
- [Coquand *et al.*, 1987] T. Coquand, C. Gunter, and G. Winskel. dI-domains as a model of polymorphism. In *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, pages 344–363. Springer-Verlag, 1987.
- [Crole and Pitts, 1992] R. Crole and A. M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and FIX-logic. *Information and Computation*, 98:171–210, 1992.
- [Curien, 1993] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhäuser, second edition, 1993.
- [Davey and Priestley, 1990] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.
- [Davies *et al.*, 1971] R. O. Davies, A. Hayes, and G. Rousseau. Complete lattices and the generalized Cantor theorem. *Proceedings of the AMS*, 27:253–258, 1971.
- [de Bakker and Scott, 1969] J. W. de Bakker and D. S. Scott. A theory of programs. Notes, IBM Seminar, Vienna, 1969.
- [Droste and Göbel, 1990] M. Droste and R. Göbel. Universal domains in the theory of denotational semantics of programming languages. In *IEEE Symposium on Logic in Computer Science*, pages 19–34. IEEE Computer Society Press, 1990.
- [Droste and Göbel, 1991] M. Droste and R. Göbel. Universal information systems. *International Journal of Foundations of Computer Science*, 1:413–424, 1991.
- [Droste and Göbel, 1993] M. Droste and R. Göbel. Universal domains and the amalgamation property. *Mathematical Structures in Computer Science*, 3:137–159, 1993.
- [Edalat, 1993a] A. Edalat. Domain theory and integration. Draft paper, 1993.
- [Edalat, 1993b] A. Edalat. Dynamical systems, measures and fractals via domain theory: extended abstract. In G. Burn, S. Gay, and M. Ryan, editors, *Theory and Formal Methods 1993*, Workshops in Computing, pages 82–99. Springer, 1993.



- [Edalat and Smyth, 1993] A. Edalat and M. B. Smyth. I-categories as a framework for solving domain equations. *Theoretical Computer Science*, 115(1):77–106, 1993.
- [Ehrhard, 1993] T. Ehrhard. Hypercoherences: A strongly stable model of linear logic. *Mathematical Structures in Computer Science*, 3(4):365–386, December 1993.
- [Ehrig and Mahr, 1985] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.
- [Erné, 1985] M. Ern  . Posets isomorphic to their extensions. *Order*, 2:199–210, 1985.
- [Ershov, 1975] Y. L. Ershov. The theory of A-spaces (English translation). *Algebra and Logic*, 12:209–232, 1975.
- [Fiech, 1992] A. Fiech. Colimits in the category CPO. Technical report, Kansas State University, 1992.
- [Freyd, 1991] P. J. Freyd. Algebraically complete categories. In A. Carboni, M. C. Pedicchio, and G. Rosolini, editors, *Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, 1991.
- [Freyd, 1992] P. J. Freyd. Remarks on algebraically compact categories. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *L.M.S. Lecture Notes*, pages 95–106. Cambridge University Press, 1992.
- [Freyd et al., 1990] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. In *Logic in Computer Science*, pages 346–354. IEEE Computer Society Press, 1990.
- [Gierz et al., 1980] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, 1980.
- [Ginsburg and Rice, 1962] S. Ginsburg and H. G. Rice. Two families of languages related to ALGOL. *Journal of the ACM*, 9:350–371, 1962.
- [Girard, 1986] J.-Y. Girard. The system  $f$  of variable types: Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.
- [Girard, 1987] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [Goguen et al., 1978] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology IV: Data Structuring*, pages 80–144. Prentice-Hall, 1978.
- [Graham, 1988] S. Graham. Closure properties of a probabilistic powerdomain construction. In M. Main, A. Melton, M. Mislove, and D. Schmidt,

- editors, *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*, pages 213–233. Springer-Verlag, 1988.
- [Gunter, 1986] C. Gunter. The largest first-order axiomatizable cartesian closed category of domains. In *Symposium on Logic in Computer Science*, pages 142–148. IEEE Computer Society Press, 1986.
- [Gunter, 1992a] C. Gunter. The mixed power domain. *Theoretical Computer Science*, 103:311–334, 1992.
- [Gunter, 1992b] C. Gunter. *Semantics of Programming Languages. Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
- [Gunter and Jung, 1988] C. Gunter and A. Jung. Coherence and consistency in domains. In *Third Annual Symposium on Logic in Computer Science*, pages 309–317. Computer Society Press of the IEEE, 1988.
- [Gunter and Scott, 1990] C. Gunter and D. S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 12: Semantic Domains, pages 633–674. Elsevier Science Publishers, Amsterdam, 1990.
- [Hausdorff, 1914] F. Hausdorff. *Grundzüge der Mengenlehre*. Veit, 1914.
- [Heckmann, 1990] R. Heckmann. *Power Domain Constructions*. PhD thesis, Universität des Saarlandes, 1990.
- [Heckmann, 1991] R. Heckmann. Power domain constructions. *Science of Computer Programming*, 17:77–117, 1991.
- [Heckmann, 1993a] R. Heckmann. Observable modules and power domain constructions. In M. Droste and Y. Gurevich, editors, *Semantics of Programming Languages and Model Theory*, volume 5 of *Algebra, Logic, and Applications*, pages 159–187. Gordon and Breach Science Publishers, 1993.
- [Heckmann, 1993b] R. Heckmann. Power domains and second order predicates. *Theoretical Computer Science*, 111:59–88, 1993.
- [Hennessy, 1993] M. Hennessy. A fully abstract denotational model for higher order processes. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 397–408. IEEE Computer Society Press, 1993.
- [Hennessy and Plotkin, 1979] M. C. B. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In J. Beçvar, editor, *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [Herrlich and Strecker, 1973] H. Herrlich and G. E. Strecker. *Category Theory*. Allyn and Bacon, 1973.
- [Hoffmann, 1981] R.-E. Hoffmann. Continuous posets, prime spectra of completely distributive complete lattices, and Hausdorff compactification. In B. Banaschewski and R.-E. Hoffmann, editors, *Continuous Lattices*, volume 871 of *Lecture Notes in Computer Science*, pages 159–208.

- Springer-Verlag, 1981.
- [Hofmann and Mislove, 1981] K. H. Hofmann and M. Mislove. Local compactness and continuous lattices. In B. Banaschewski and R.-E. Hoffmann, editors, *Continuous Lattices, Proceedings Bremen 1979*, volume 871 of *Lecture Notes in Mathematics*, pages 209–248. Springer-Verlag, 1981.
- [Hofmann and Mislove, 1993] K. H. Hofmann and M. W. Mislove. All compact Hausdorff lambda models are degenerate. Technical Report 93–1, Tulane University, 1993.
- [Hoofman, 1992] R. Hoofman. *Non-stable models of linear logic*. PhD thesis, University of Utrecht, 1992.
- [Hosono and Sato, 1977] Ch. Hosono and M. Sato. The retracts in  $\mathcal{P}\omega$  do not form a continuous lattice—a solution to Scott’s problem. *Theoretical Computer Science*, 4:137–142, 1977.
- [Hrbacek, 1987] K. Hrbacek. Convex powerdomains I. *Information and Computation*, 74:198–225, 1987.
- [Hrbacek, 1988] K. Hrbacek. A powerdomain construction. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 298 of *Lecture Notes in Computer Science*, pages 200–212. Springer-Verlag, 1988.
- [Hrbacek, 1989] K. Hrbacek. Convex powerdomains II. *Information and Computation*, 81:290–317, 1989.
- [Huth, 1992] M. Huth. Cartesian closed categories of domains and the space  $\text{Proj}(D)$ . In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 259–271. Springer-Verlag, 1992.
- [Huth, 1994] M. Huth. Linear domains and linear maps. In S. Brookes, M. Main, A. Melton, M. Mislove and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pp. 438–453, *Lecture Notes in Computer Science*, Vol. 802. Springer-Verlag, 1994. to appear.
- [Huwig and Poigné, 1990] H. Huwig and A. Poigné. A note on inconsistencies caused by fixpoints in a cartesian closed category. *Theoretical Computer Science*, 73:101–112, 1990.
- [Hyland, 1991] J. M. E. Hyland. First steps in synthetic domain theory. In A. Carboni, C. Pedicchio, and G. Rosolini, editors, *Conference on Category Theory 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 131–156. Springer-Verlag, 1991.
- [Isbell, 1972] J. Isbell. Atomless parts of spaces. *Mathematica Scandinavica*, 31:5–32, 1972.
- [Isbell, 1982] J. Isbell. Completion of a construction of Johnstone. *Proceedings of the American Mathematical Society*, 85:333–334, 1982.

- [Iwamura, 1944] T. Iwamura. A lemma on directed sets. *Zenkoku Shijo Sugaku Danwakai*, 262:107–111, 1944. (In Japanese).
- [Jensen, 1991] T. Jensen. Strictness analysis in logical form. In J. Hughes, editor, *Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*. Springer Verlag, 1991.
- [Jensen, 1992] T. Jensen. Disjunctive strictness analysis. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 174–185. IEEE Computer Science Press, 1992.
- [Johnstone, 1981] P. T. Johnstone. Scott is not always sober. In B. Banaschewski and R.-E. Hoffmann, editors, *Continuous Lattices*, volume 871 of *Lecture Notes in Mathematics*, pages 282–283. Springer-Verlag, 1981.
- [Johnstone, 1982] P. T. Johnstone. *Stone Spaces*, volume 3 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1982.
- [Johnstone, 1983] P. T. Johnstone. The point of pointless topology. *Bulletin of the American Mathematical Society*, 8:41–53, 1983.
- [Jones, 1990] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, Edinburgh, 1990. Also published as Technical Report No. CST-63-90.
- [Jones and Plotkin, 1989] C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science*, pages 186–195. IEEE Computer Society Press, 1989.
- [Jung, 1988] A. Jung. New results on hierarchies of domains. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*, pages 303–310. Springer-Verlag, 1988.
- [Jung, 1989] A. Jung. *Cartesian Closed Categories of Domains*, volume 66 of *CWI Tracts*. Centrum voor Wiskunde en Informatica, Amsterdam, 1989.
- [Jung, 1990] A. Jung. The classification of continuous domains. In *Logic in Computer Science*, pages 35–40. IEEE Computer Society Press, 1990.
- [Kahn and Plotkin, 1993] G. Kahn and G. Plotkin. Concrete domains. *Theoretical Computer Science*, 121:187–277, 1993. Translation of [Kahn and Plotkin, 1993].
- [Kanda, 1979] A. Kanda. Fully effective solutions of recursive domain equations. In J. Bečvar, editor, *Mathematical Foundations of Computer Science*, volume 74. Springer-Verlag, 1979. *Lecture Notes in Computer Science*.
- [Keimel and Paseka, to appear] K. Keimel and J. Paseka. A direct proof of the Hofmann-Mislove theorem. *Proceedings of the AMS*, Vol. 120, pp. 301–303, 1994.



- [Kleene, 1952] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
- [Kock, 1970] A. Kock. Monads on symmetric monoidal closed categories. *Archiv der Mathematik*, 21:1–10, 1970.
- [Kock, 1972] A. Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23:113–120, 1972.
- [Kock, 1981] A. Kock. *Synthetic Differential Geometry*, volume 51 of *LMS Lecture Notes*. Cambridge University Press, 1981.
- [Krasner, 1939] M. Krasner. Un type d'ensembles semi-ordonnés et ses rapports avec une hypothèse de M. A. Weil. *Bulletin de la Société Mathématique de France*, 67:162–176, 1939.
- [Lassez et al., 1982] J.-L. Lassez, V. L. Nguyen, and E. A. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14:112–116, 1982.
- [Lawson, 1979] J. Lawson. The duality of continuous posets. *Houston Journal of Mathematics*, 5:357–394, 1979.
- [Lehmann and Smyth, 1981] D. J. Lehmann and M. B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.
- [McCarty, 1984] D. C. McCarty. *Realizability and Recursive Mathematics*. PhD thesis, Oxford University, 1984.
- [MacQueen et al., 1986] D. MacQueen, G. D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.
- [Main, 1985] M. Main. Free constructions of powerdomains. In A. Melton, editor, *Mathematical Foundations of Programming Semantics*, volume 239 of *Lecture Notes in Computer Science*, pages 162–183. Springer-Verlag, 1985.
- [Markowsky, 1976] G. Markowsky. Chain-complete p.o. sets and directed sets with applications. *Algebra Universalis*, 6:53–68, 1976.
- [Markowsky, 1977] G. Markowsky. Categories of chain-complete posets. *Theoretical Computer Science*, 4:125–135, 1977.
- [Markowsky, 1981] G. Markowsky. A motivation and generalization of Scott's notion of a continuous lattice. In B. Banaschewski and R.-E. Hoffmann, editors, *Continuous Lattices*, volume 871 of *Lecture Notes in Mathematics*, pages 298–307. Springer-Verlag, 1981.
- [Meseguer, 1977] J. Meseguer. On order-complete universal algebra and enriched functorial semantics. In M. Karpinski, editor, *Fundamentals of Computation Theory*, volume 56 of *Lecture Notes in Computer Science*, pages 294–301. Springer-Verlag, 1977.
- [Milne and Strachey, 1976] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1976.



- [Milner and Tofte, 1991] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.
- [Milner et al., 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [Moggi, 1991] E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.
- [Nelson, 1981] E. Nelson. Free Z-continuous algebras. In B. Banaschewski and R. E. Hoffmann, editors, *Continuous Lattices*, volume 871 of *Lecture Notes in Mathematics*, pages 315–334. Springer-Verlag, 1981.
- [Nivat and Reynolds, 1985] M. Nivat and J. C. Reynolds, editors. *Algebraic Methods in Computer Science*. Cambridge University Press, 1985.
- [Nüßler, 1992] K.-J. Nüßler. *Ordnungstheoretische Modelle für nicht-deterministische Programmiersprachen*. PhD thesis, Universität GH Essen, 1992.
- [Ong, 1993] C.-H. L. Ong. Nondeterminism in a functional setting. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 275–286. IEEE Computer Society Press, 1993.
- [Park, 1969] D. M. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 59–78. Edinburgh University Press, 1969.
- [Pereira and Shieber, 1984] F. Pereira and S. Shieber. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 123–129. Association for Computational Linguistics, 1984.
- [Phoa, 1991] W. Phoa. *Domain Theory in Realizability Toposes*. PhD thesis, University of Cambridge, 1991.
- [Pitts, 1992] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.
- [Pitts, 1993a] A. M. Pitts. Relational properties of recursively defined domains. In *Logic in Computer Science*, pages 86–97. IEEE Computer Society Press, 1993.
- [Pitts, 1993b] A. M. Pitts. Relational properties of recursively defined domains. Technical Report 321, Cambridge University, 1993.
- [Platek, 1964] R. Platek. *New foundations for recursion theory*. PhD thesis, Stanford University, 1964.
- [Plotkin, 1976] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452–487, 1976.
- [Plotkin, 1981] G. D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the “Pisa Notes”). Dept. of Computer Science, University of Edinburgh, 1981.
- [Plotkin, 1985] G. D. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language

and Information, Stanford, 1985.

- [Plotkin, 1993] G. D. Plotkin. Type theory and recursion. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, page 374. IEEE Computer Society Press, 1993.
- [Poigné, 1992] A. Poigné. Basic category theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 1, pages 413–640. Oxford University Press, 1992.
- [Pollard and Moshier, 1990] C. Pollard and D. Moshier. Unifying partial descriptions of sets. In P. Hanson, editor, *Information, Language and Cognition*, volume 1 of *Vancouver Studies in Cognitive Science*. University of British Columbia Press, 1990.
- [Puhlmann, 1993] H. Puhlmann. The snack powerdomain for database semantics. In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 650–659. Springer-Verlag, 1993.
- [Raney, 1953] G. N. Raney. A subdirect-union representation for completely distributive complete lattices. *Proc. AMS*, 4:518–522, 1953.
- [Raney, 1960] G. N. Raney. Tight Galois connections and complete distributivity. *Trans. AMS*, 97:418–426, 1960.
- [Robinson and Rosolini, 1988] E. Robinson and G. Rosolini. Categories of partial maps. *Information and Computation*, 79:95–130, 1988.
- [Rosolini, 1986] G. Rosolini. *Continuity and Effectiveness in Topoi*. PhD thesis, Oxford University, 1986.
- [Rounds and Zhang, 1994] W. Rounds and G. Q. Zhang. Domain theory meets default logic. *Journal of Logic and Computation*, 4:1–24, 1994.
- [Saheb-Djahromi, 1980] N. Saheb-Djahromi. CPO's of measures for non-determinism. *Theoretical Computer Science*, 12:19–37, 1980.
- [Schalk, 1993] A. Schalk. *Algebras for Generalized Power Constructions*. Doctoral thesis, Technische Hochschule Darmstadt, 1993. 174 pp.
- [Schmidt, 1986] D. A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.
- [Scott, 1969] D. S. Scott. A type theoretic alternative to ISWIM, CUCH, OWHY. Manuscript, University of Oxford, 1969.
- [Scott, 1970] D. S. Scott. Outline of a mathematical theory of computation. In *4th Annual Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.
- [Scott, 1971] D. S. Scott. The lattice of flow diagrams. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 311–366. Springer-Verlag, 1971.
- [Scott, 1972] D. S. Scott. Continuous lattices. In E. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136. Springer-Verlag, Berlin, 1972. Lecture Note in Mathematics 274.

- [Scott, 1976] D. S. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.
- [Scott, 1980] D. S. Scott. Relating theories of lambda calculus. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450. Academic Press, 1980.
- [Scott, 1981] D. S. Scott. Lectures on a mathematical theory of computation. Monograph PRG-19, Oxford University Computing Laboratory, Oxford, 1981.
- [Scott, 1982] D. S. Scott. Domains for denotational semantics. In M. Nielson and E. M. Schmidt, editors, *International Colloquium on Automata, Languages and Programs*, volume 140, pages 577–613, Berlin, 1982. Springer-Verlag.
- [Scott, 1993] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Reprint of a manuscript written in 1969.
- [Smyth, 1977] M. B. Smyth. Effectively given domains. *Theoretical Computer Science*, 5:257–274, 1977.
- [Smyth, 1978] M. B. Smyth. Powerdomains. *Journal of Computer and Systems Sciences*, 16:23–36, 1978.
- [Smyth, 1983a] M. B. Smyth. The largest cartesian closed category of domains. *Theoretical Computer Science*, 27:109–119, 1983.
- [Smyth, 1983b] M. B. Smyth. Powerdomains and predicate transformers: a topological view. In J. Diaz, editor, *Automata, Languages and Programming*, pages 662–675, Berlin, 1983. Springer-Verlag. Lecture Notes in Computer Science Vol. 154.
- [Smyth, 1986] M. B. Smyth. Finite approximation of spaces. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 225–241. Springer-Verlag, 1986.
- [Smyth, 1992] M. B. Smyth. Topology. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 1, pages 641–761. Oxford University Press, 1992.
- [Smyth and Plotkin, 1982] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing*, 11:761–783, 1982.
- [Stoughton, 1988] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman/Wiley, 1988.
- [Stoy, 1977] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

- [Taylor, 1987] P. Taylor. *Recursive Domains, Indexed Category Theory and Polymorphism*. PhD thesis, Cambridge University, 1987.
- [Taylor, 1990] P. Taylor. An algebraic approach to stable domains. *Pure and Applied Algebra*, 64, 1990.
- [Taylor, 1991] P. Taylor. The fixed point property in synthetic domain theory. In *6th LICS conference*, pages 152–160. IEEE Computer Society Press, 1991.
- [Tennent, 1991] R. D. Tennent. *Semantics of Programming Languages*. Prentice-Hall, 1991.
- [Tischendorf and Tůma, 1993] M. Tischendorf and J. Tůma. The characterization of congruence lattices of lattices. Technical Report 1559, Technische Hochschule Darmstadt, 1993.
- [Vickers, 1989] S. J. Vickers. *Topology Via Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
- [Vietoris, 1921] L. Vietoris. Stetige Mengen. *Monatshefte für Mathematik und Physik*, 31:173–204, 1921.
- [Vietoris, 1922] L. Vietoris. Bereiche zweiter Ordnung. *Monatshefte für Mathematik und Physik*, 32:258–280, 1922.
- [Weihrauch and Deil, 1980] K. Weihrauch and T. Deil. Berechenbarkeit auf cpo's. Technical Report 63, Rheinisch-Westfälische Technische Hochschule Aachen, 1980.
- [Winskel, 1983] G. Winskel. Powerdomains and modality. In M. Karpinski, editor, *Foundations of Computation Theory*, pages 505–514. Springer-Verlag, Berlin, 1983. Lecture Notes in Computer Science Vol. 158.
- [Winskel, 1988] G. Winskel. An introduction to event structures. In J. W. de Bakker, editor, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 364–399. Springer-Verlag, 1988.
- [Winskel, 1993] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. MIT Press, 1993.
- [Zhang, 1991] G.-Q. Zhang. *Logic of Domains*. Progress in Theoretical Computer Science. Birkhäuser, 1991.

# Denotational Semantics

R. D. Tennent

---

## Contents

1	Introduction . . . . .	170
	1.1 Approaches . . . . .	171
	1.2 An example: binary numerals . . . . .	172
	1.3 Compositionality . . . . .	175
	1.4 Criteria . . . . .	176
	1.5 Overview . . . . .	178
	1.6 Bibliographic notes . . . . .	179
2	A simple imperative language . . . . .	180
	2.1 Expressions and commands . . . . .	180
	2.2 Assignment Commands . . . . .	185
	2.3 Indefinite iterations . . . . .	187
	2.4 Programs . . . . .	191
	2.5 Operational semantics . . . . .	192
	2.6 Programming logic . . . . .	196
	2.7 Non-determinism . . . . .	203
	2.8 Bibliographic notes . . . . .	205
3	A simple applicative language . . . . .	205
	3.1 Definitions and function applications . . . . .	206
	3.2 Function definitions . . . . .	212
	3.3 Defined notation . . . . .	214
	3.4 Elementary properties . . . . .	217
	3.5 Programming logic . . . . .	220
	3.6 Bibliographic notes . . . . .	228
4	Recursion . . . . .	228
	4.1 Recursive definitions . . . . .	228
	4.2 Domain-theoretic semantics . . . . .	231
	4.3 Operational semantics . . . . .	237
	4.4 Programming logic . . . . .	239
	4.5 Full abstraction . . . . .	242
	4.6 Untyped procedures . . . . .	243
	4.7 Bibliographic notes . . . . .	245



5	An Algol-like language I . . . . .	245
	5.1 Syntax . . . . .	246
	5.2 Semantics . . . . .	250
	5.3 Call by value . . . . .	253
	5.4 Programming logic . . . . .	256
	5.5 Bibliographic notes . . . . .	260
6	An Algol-like language II . . . . .	260
	6.1 Coercions . . . . .	261
	6.2 Local variables . . . . .	268
	6.3 Product types and arrays . . . . .	270
	6.4 Lists . . . . .	274
	6.5 Acceptors . . . . .	279
	6.6 Jumps . . . . .	282
	6.7 Intermediate output . . . . .	287
	6.8 Block expressions . . . . .	288
	6.9 Bibliographic notes . . . . .	289
7	Possible worlds . . . . .	290
	7.1 Functor-category semantics . . . . .	291
	7.2 Semantic-domain functors . . . . .	292
	7.3 Semantic valuations . . . . .	295
	7.4 Semantics of local variables . . . . .	298
	7.5 Specifications . . . . .	302
	7.6 Non-interference specifications . . . . .	304
	7.7 Semantics of block expressions . . . . .	308
	7.8 Bibliographic notes . . . . .	311

## 1 Introduction

Designers, implementers, and serious users of a programming language need a complete and accurate understanding of the *semantics* (the intended meaning) of every construct of that language. The semantic descriptions in reference manuals and language standards are almost always inadequate because they are based primarily on implementation techniques and intuition. A rigorous mathematical theory of the semantics of programming languages is needed to support correct description and implementation of their meanings, systematic development and verification of programs, analysis of existing programming languages, and design of new languages that are simpler and more regular.

## 1.1 Approaches

### 1.1.1 Operational semantics

The meanings of programs can be described in terms of sequences of computational steps on an abstract or idealized computer. This has been termed the *operational* approach to semantics. Operational descriptions can be useful to implementers by providing precise formulations of implementation techniques.

However, such semantic descriptions can be very intricate, and the concepts underlying the linguistic features can be so obscured by representational detail that the *correctness* of the description may be in doubt. Furthermore, it can be very difficult to show *equivalence* of different styles of implementation of a language without a representation-independent standard to which they can be related.

### 1.1.2 Axiomatic semantics

A more abstract approach is to adopt a formal system for reasoning about program meanings as an *implicit* description of the semantics of the language. This has come to be known as the *axiomatic* approach to semantics. A programmer can use such a formal system to help develop and verify programs, and a language designer can use its complexity as one measure of the success of the language design.

However, it is important that such a formal system be *sound*; i.e., that every formula that can be derived in the formal system be true in the intended interpretation. It has happened that ‘axiomatic definitions’ of programming languages subsequently turned out to be incorrect or even inconsistent. The converse question of *completeness* is also of theoretical interest: is *every* true formula derivable in the formal system? To address such questions, a different kind of semantic description must be used.

### 1.1.3 Denotational semantics

For mathematical logicians, a semantic interpretation of a formal language is essentially a *valuation function* mapping every well-formed phrase into its intended meaning in an appropriate domain of mathematical entities, such as numbers, truth values, or functions. It is conventional in logic to define such valuations in a ‘compositional’ manner; that is, the meaning of every composite phrase is expressed as a function of the meanings of its immediate sub-phrases. This is the property that characterizes *denotational* semantics. In this chapter, the term ‘semantics’ (without further qualification) will be used in the logician’s sense of *denotational* semantics.

## 1.2 An example: binary numerals

In this section we present a description of the syntax and (denotational) semantics of the ‘language’ of binary numerals; i.e., positional representation of natural numbers, using base 2. For example, 1101 is a binary numeral and it denotes the natural number thirteen.

The first step is to name the *phrase types* for the language, as follows:

**bin**    binary-number phrases  
**nat**    natural-number phrases,

and define the corresponding *domains of interpretation*, or sets of possible meanings:

$$\begin{aligned} \llbracket \text{bin} \rrbracket &= \{0, 1\} \\ \llbracket \text{nat} \rrbracket &= \{0, 1, 2, \dots\} \end{aligned}$$

For any phrase type  $\theta$ ,  $\llbracket \theta \rrbracket$  will always denote the corresponding domain of interpretation. Notice that  $\llbracket \text{bin} \rrbracket$ , the set of binary numbers, is a subset of  $\llbracket \text{nat} \rrbracket$ , the set of all natural numbers.

Next, we specify the *syntax* of the language of binary numerals. It is conventional in computer science to use a notation known as BNF (Backus–Naur Formalism) to describe syntax formally, as in the following:

$$\begin{aligned} \langle \text{bin} \rangle &::= 0 \mid 1 \\ \langle \text{nat} \rangle &::= \langle \text{bin} \rangle \mid \langle \text{nat} \rangle \langle \text{bin} \rangle \end{aligned}$$

Intuitively, this states that

- (i) 0 is a binary-number phrase;
- (ii) 1 is a binary-number phrase;
- (iii) if  $B$  is any binary-number phrase then  $B$  is also a natural-number phrase;
- (iv) if  $N$  is any natural-number phrase and  $B$  is any binary-number phrase then  $NB$  is a natural-number phrase,

and there are no other binary-number or natural-number phrases.

For more complex languages, BNF and similar context-free formalisms are inadequate, and in this chapter we will use another meta-notation to describe syntax: formal systems of axioms and inference rules for formulas of the form  $X: \theta$ , interpreted as asserting that  $X$  is a well-formed phrase of type  $\theta$ . A description of the syntax of binary numerals in this notation is given in Table 1.

The first two rules have an empty set of premisses above the horizontal line, and so are *axioms*. The remaining two rules have one or two premisses, and are *inference rules*. The symbols  $B$  and  $N$  used in the rules are termed (syntactic) *meta-variables*. In our meta-language (i.e., the language in which the description is written), they stand for arbitrary binary-number

Zero:

$$\frac{}{0: \text{bin}}$$

One:

$$\frac{}{1: \text{bin}}$$

Primitive Numeral:

$$\frac{B: \text{bin}}{B: \text{nat}}$$

Composite Numeral:

$$\frac{N: \text{nat} \quad B: \text{bin}}{NB: \text{nat}}$$

**Table 1.** Syntax of binary numerals

and natural-number phrases, respectively, of the object language (i.e., the language being described). The conclusions of rules are arbitrary syntactic phrases, but the meta-variables in the conclusion of each rule are those whose types are defined in the premisses of the rule, so that the general form of an inference rule is

$$\frac{\dots X_i: \theta_i \dots}{\dots X_i \dots: \theta}$$

Here is a simple derivation using the formal system of Table 1:

$$\frac{\frac{\frac{1: \text{bin}}{1: \text{nat}} \quad \frac{1: \text{bin}}{1: \text{bin}}}{11: \text{nat}} \quad \frac{0: \text{bin}}{0: \text{bin}}}{\frac{110: \text{nat}}{110: \text{nat}} \quad \frac{1: \text{bin}}{1: \text{bin}}}{1101: \text{nat}}$$

The derivation defines the ‘phrase structure’ of the numeral 1101.

Finally, we complete the semantic description by defining the following *valuation functions*:

$$\begin{aligned} \llbracket \cdot \rrbracket_{\text{bin}}: [\text{bin}] &\rightarrow \llbracket \text{bin} \rrbracket \\ \llbracket \cdot \rrbracket_{\text{nat}}: [\text{nat}] &\rightarrow \llbracket \text{nat} \rrbracket \end{aligned}$$

where, for any phrase type  $\theta$ ,  $[\theta]$  denotes the set of all well-formed phrases of type  $\theta$ . These functions will map syntactically well-formed phrases to their intended meanings in the appropriate domain of interpretation. That is, for any phrase  $X \in [\theta]$ ,  $\llbracket X \rrbracket_\theta \in \llbracket \theta \rrbracket$  is the intended meaning of  $X$ ; for example, we want  $\llbracket 1101 \rrbracket_{\text{nat}} = 13$ . It is appropriate that the semantics be given by *functions*: we want *every* well-formed phrase to have a *unique* meaning.

The type subscript on phrase-valuation brackets will often be omitted when the type is obvious from the phrase itself or from the context. In

many presentations, valuation functions are written  $\mathcal{B}[\cdot]$ ,  $\mathcal{N}[\cdot]$ , and so on, where here the double brackets help separate the language being defined from the meta-language. Other authors provide typing information *within* the brackets, as in  $\llbracket N : \text{nat} \rrbracket$ .

The valuation functions for our language can be defined by the ‘semantic equations’ in Table 2.

$$\begin{aligned}\llbracket 0 \rrbracket_{\text{bin}} &= 0 \\ \llbracket 1 \rrbracket_{\text{bin}} &= 1 \\ \llbracket B \rrbracket_{\text{nat}} &= \llbracket B \rrbracket_{\text{bin}} \\ \llbracket NB \rrbracket_{\text{nat}} &= 2 \times \llbracket N \rrbracket_{\text{nat}} + \llbracket B \rrbracket_{\text{bin}}\end{aligned}$$

**Table 2.** Semantic equations for binary numerals

Notice that there is one semantic equation for each rule in the syntax, and that the syntactic meta-variables in each equation are those of the corresponding syntax rule.

It is not entirely obvious that these equations properly define mathematical functions. This is clear for  $\llbracket \cdot \rrbracket_{\text{bin}}$ , but notice that  $\llbracket \cdot \rrbracket_{\text{nat}}$  appears on *both* sides of the last of the equations! However, the use on the right-hand side is on a *sub*-phrase  $N$  of the composite phrase  $NB$  on the left-hand side, and so  $\llbracket N \rrbracket_{\text{nat}}$  is *uniquely* defined for *every* binary numeral  $N$  because the (unique) syntactic derivation for any binary numeral is of finite length. For example, we can determine the meaning of the numeral 1011 as follows:

$$\begin{aligned}\llbracket 1101 \rrbracket_{\text{nat}} &= 2 \times \llbracket 110 \rrbracket_{\text{nat}} + 1 \\ &= 2 \times (2 \times \llbracket 11 \rrbracket_{\text{nat}} + 0) + 1 \\ &= 2 \times (2 \times (2 \times \llbracket 1 \rrbracket_{\text{nat}} + 1) + 0) + 1 \\ &= 2 \times (2 \times (2 \times 1 + 1) + 0) + 1 \\ &= 13.\end{aligned}$$

A sceptical reader might also question whether an equation such as  $\llbracket 0 \rrbracket_{\text{bin}} = 0$  has any content. Despite appearances, it is not circular. The 0 on the left-hand side is in the *object language*; it is part of the notation chosen by the designer of that language. The 0 on the right-hand side represents the natural number zero in the *meta-language*, the language we use to communicate the syntactic and semantic descriptions. We could, if we chose to, adopt any other acceptable meta-linguistic representation of the natural number zero without significantly changing the semantic valuation; but we could not change the 0 in the object language without thereby changing the language we were attempting to describe.

The equation  $\llbracket 0 \rrbracket_{\text{bin}} = 0$  simply states that the object-language symbol 0 denotes the natural number zero. It is just one part of a description that, as a whole, is definitely non-trivial: it explicates the *linguistic* concept of



positional notation in terms of the *mathematical* operations of multiplication and addition of natural numbers, which are independent of any specific representation of the natural numbers.

**Exercise 1.2.1.** Describe the syntax and denotational semantics of ‘reduced’ binary numerals (i.e., without unnecessary leading zeroes); for example, the numeral 001101 is not reduced, but 1101 and 0 are. Hint: use a phrase type **pos** (positive numbers) with  $\llbracket \mathbf{pos} \rrbracket = \{1, 2, 3, \dots\}$ .

**Exercise 1.2.2.** Describe the syntax and denotational semantics of binary numerals with fractions (e.g., 101.1011, which denotes the rational number 5.6875). You may assume that  $\mathbb{Q}$  denotes the set of all rational numbers.

### 1.3 Compositionality

If  $X$  and  $X'$  are well-formed phrases of some type  $\theta$ , we say that  $X$  and  $X'$  are *semantically equivalent* and write  $X \equiv_{\theta} X'$  when  $\llbracket X \rrbracket_{\theta} = \llbracket X' \rrbracket_{\theta}$ . An important benefit of compositionality is that replacing any component of a program by a semantically equivalent phrase yields a program that is semantically equivalent to the original; that is,  $X \equiv_{\theta} X'$  implies that, for every program context  $\dots \_ \dots$  (of type  $\theta'$ , say) in which phrases of type  $\theta$  may be used,

$$\dots X \dots \equiv_{\theta'} \dots X' \dots$$

For example, it is evident from the semantics of the preceding section that  $0B \equiv_{\mathbf{nat}} B$  for all  $B$ : **bin**, and this means that *any* numeral with a ‘leading zero’ can be simplified to a semantically equivalent numeral without that leading zero. This illustrates how semantic equivalences can be used to justify optimizing code transformations by compilers.

A technical benefit of compositionality is that it allows properties of valuations to be proved by *structural induction*, that is, induction on the syntactic complexity of linguistic phrases: a property is proved

- (i) for each of the syntactic primitives (the basis), and
- (ii) for each of the composite constructs, on the assumption (the inductive hypothesis) that its immediate constituents have the property.

The conclusion is that all phrases have the property.

For example, consider the ‘successor’ operation  $'$  on binary numerals defined as follows:

$$\begin{aligned} 0' &= 1 \\ 1' &= 10 \\ N0' &= N1 \\ N1' &= N'0 \end{aligned}$$

We use structural induction to prove the correctness of this definition.

**Proposition 1.3.1.** *For all  $N \in [\mathbf{nat}]$ ,  $\llbracket N' \rrbracket = \llbracket N \rrbracket + 1$ .*

**Proof.** Cases 0, 1, and  $N0$  are immediate; for numerals of the form  $N1$ :

$$\begin{aligned} \llbracket N1' \rrbracket &= \llbracket N'0 \rrbracket \\ &= 2 \times \llbracket N' \rrbracket \\ &= 2 \times (\llbracket N \rrbracket + 1) \quad (\text{by induction}) \\ &= \llbracket N1 \rrbracket + 1. \end{aligned}$$

We conclude that, for *all*  $N \in [\mathbf{nat}]$ ,  $\llbracket N' \rrbracket = \llbracket N \rrbracket + 1$ . ■

**Exercise 1.3.2.** Define an operation  $\oplus$  on binary numerals such that

$$\llbracket N_0 \oplus N_1 \rrbracket = \llbracket N_0 \rrbracket + \llbracket N_1 \rrbracket$$

and verify the correctness of the definition.

It is generally thought that compositionality also has conceptual benefits, particularly to a language designer: each feature of a language can be considered and evaluated in relative isolation from other linguistic elements, and the semantic interpretation is *structured* by the syntax of the language.

As an example of a semantical treatment that is *not* compositional, consider the equation

$$\llbracket \mathbf{while} \ E \ \mathbf{do} \ C \rrbracket = \llbracket \mathbf{if} \ E \ \mathbf{then} \ (C ; \mathbf{while} \ E \ \mathbf{do} \ C) \rrbracket.$$

Although one would expect this equation to be *true* of the meanings, it is not a compositional interpretation of the **while** loop, because the phrase on the right-hand side is not a proper sub-phrase of that on the left.

As another example, consider interpreting procedure definitions in terms of the *text* of the definition or a translation of it, as is done by a compiler or interpreter. The valuation for procedure *calls* must then be expressed in terms of the meaning of that text; however, the text defining the procedure is not usually a sub-phrase of the call, so this would not be a compositional interpretation.

## 1.4 Criteria

Ideally, rigorous semantic descriptions of programming languages would be comprehensible to every programmer, implementer, and language designer; but it has become apparent that the necessary mathematics is too challenging for this goal to be achievable. Fortunately, it is *not* essential that programmers and implementers understand rigorous semantics directly. There are other ways to express the aspects of programming-language semantics

that are relevant to programmers and implementers. Their needs can be satisfied by

- informal descriptions, provided these are accurate because they are based on rigorous semantics;
- formal systems for reasoning about programs, provided these have been rigorously proved sound; and
- formal descriptions of implementation techniques, provided these have been rigorously verified.

This means that informal, operational, axiomatic, and denotational approaches to semantics are *all* useful in various ways and complement each other. A theory of semantics should not be judged by its comprehensibility to programmers and implementers, but rather by the extent to which it can provide rigorous justifications for techniques usable by them. A comparable situation is that engineers rarely need to understand the foundational constructions of the set of real numbers using Cauchy sequences or Dedekind cuts.

It is also unrealistic to expect a semantic theory to provide elegant descriptions of existing ‘practical’ programming languages. Most languages used in practice are rather irregular and have been compromised in various ways by pragmatic considerations. Detailed study of such languages was necessary in the early stages of the development of the theory before the conceptual issues were understood. But it has become increasingly possible to focus on idealized languages that are much simpler than languages used in practice, yet are conceptually comparable to them.

A final criterion we shall mention is that it is desirable that a semantic interpretation be *fully abstract*, that is to say, that it not distinguish phrases semantically unless there exist program contexts for those phrases that allow the differences to be observed. More formally: a semantic valuation is termed *fully abstract* if and only if, for all phrase types  $\theta$  and  $P, P' \in [\theta]$ , if  $\dots P \dots \equiv_{\theta'} \dots P' \dots$  for every program  $\dots \_ \dots$  in which a phrase of type  $\theta$  may be used, then  $P \equiv_{\theta} P'$ . This is the converse of the substitutivity principle for compositional valuations discussed at the beginning of Section 1.3.

It is easy to see that the semantic interpretation of binary numerals in Section 1.2 is fully abstract. There are only two phrases 0 and 1 of type **bin** and there exist **nat** contexts that require that their values be distinct; for example,  $\llbracket 10 \rrbracket \neq \llbracket 11 \rrbracket$ , and so  $1\_$  is such a context. Unfortunately, the ideal of full abstraction is very difficult or impossible to achieve for more complex languages.

The practical significance of full abstraction is as follows: if the semantics unnecessarily distinguishes the meanings of phrases  $P$  and  $P'$ , the formal equivalence of  $P$  and  $P'$  could not be validated by the semantics.

Also, an axiom asserting that  $P$  and  $P'$  are *not* equivalent might incorrectly be regarded as sound.

In summary, a theory of programming-language semantics should aim to

- express meanings rigorously, while avoiding unnecessary complexity or implementation bias as much as possible;
- validate methods of reasoning about programs;
- verify implementation methods.

## 1.5 Overview

The primary aim of this chapter is to present the techniques that have been developed to describe ‘conventional’ programming languages denotationally and to reason about these descriptions. A second aim is to demonstrate the applicability of these techniques to verification of implementation methods and formal systems for reasoning about programs, and to programming-language design.

It is assumed that the reader has some experience with programming languages and is familiar with basic concepts in mathematical logic, domain theory, lambda calculus, and category theory, such as may be found presented in earlier chapters of this Handbook.

Rather than attempt to discuss every feature of every programming language, the chapter presents detailed studies of a small number of languages that are particularly important. The presentation starts with two significant ‘theoretical’ languages: the simple imperative language of **while** programs, and a ‘sugared’ version of the applicative language that G. D. Plotkin [1977] termed PCF; i.e., the simply-typed lambda notation augmented by recursion and arithmetic. Then, these fragments are combined and extended to encompass most of the features of conventional practical programming languages in the way that J. C. Reynolds [1981b] has characterized as *Algol 60-like*.

This organization emphasizes languages that are statically typed and deterministic, but there is also some discussion of the semantics of dynamically typed and indeterminate languages.

Readers familiar with the subject will find the presentation to be unconventional in several respects.

- Partial functions on states are used to interpret the simple imperative language and conventional set-theoretic functions are used to interpret the simple applicative language without recursion. This has made it possible to defer the introduction of domain-theoretic concepts until the reader has become comfortable with basic semantic methods using familiar mathematical tools.



- In their pioneering papers on denotational semantics of programming languages, it was convenient for Scott and Strachey to demonstrate their new techniques on typeless or dynamically typed languages. Most workers in the field (including this author) went on to use virtually the same techniques on statically typed languages, but such descriptions are unnecessarily cumbersome and unrealistic because of all the branching on types.

The presentation here uses minor variants of the traditional techniques in order to deal more satisfactorily with statically typed languages. For example, Boolean and numerical expressions are distinguished syntactically, and there are separate domains of expressible values for each of these types of expressions, rather than a single domain of *all* expressible values. Although the syntactic descriptions must be slightly more complex to make the appropriate type distinctions, the semantic valuations are much simpler and more realistic. Inference rules in natural-deduction style are used to specify non-context-free type and scope constraints on the syntax.

- It has been recognized for some time that the traditional denotational-semantic approach to dynamic storage (locations and marked stores) is not very satisfactory for describing ‘local’ (stack-implementable) storage management. In this presentation, the use of locations has been de-emphasized by using variable names in lieu of locations for as long as possible. A relatively new solution to the problem of local-variable declarations, ‘possible worlds’, is discussed in Section 7.

Most of the material in this chapter also appears in *Semantics of Programming Languages* (Prentice-Hall International, 1991). The author is very grateful to everyone who gave him comments on draft versions of this material, especially David Andrews, Ellen Attack, David Barnard, Ian Carmichael, Gwen Clarke, Laurie Hendren, David Naumann, Peter O’Hearn, Andy Pitts, John Reynolds, Edmund Robinson, David Schmidt, Allen Stoughton, Kevin Tobin, and Chris Wadsworth.

## 1.6 Bibliographic notes

Two early operational approaches to semantics are described in [Landin, 1964; Lucas and Walk, 1969]. [Reynolds, 1972; Plotkin, 1981; Hennessy, 1990] discuss properties of and relations between various operational approaches.

The axiomatic approach to semantics has its origins in [Floyd, 1967; Hoare, 1969]. The success of the approach as a method of specifying semantics is assessed in [Schwarz, 1974; Greif and Meyer, 1981; Bergstra *et al.*, 1982; Meyer and Halpern, 1982; Leivant, 1985b; Meyer, 1986].

The denotational approach to semantics may be traced back to [Frege, 1892; Carnap, 1947; Tarski, 1956]. The feasibility of describing *programming* languages compositionally was first demonstrated by D. Scott and C. Strachey [Scott and Strachey, 1971; Scott, 1972a; Strachey, 1972]. [Stoy, 1977; Gordon, 1979;



Schmidt, 1986; Manes and Arbib, 1986] are textbooks on the denotational approach to describing the semantics of programming languages and its applications. [Tennent, 1981] analyses programming languages using denotational-semantic concepts. For further discussion of compositionality, see [Janssen, 1986].

The view that the various approaches to semantics should be regarded as *complementary* seems to have originated in [Hoare and Lauer, 1974]. [Plotkin, 1975; Milne and Strachey, 1976; de Bakker *et al.*, 1980; Loeckx and Sieber, 1984; Nielson and Nielson, 1992; Gunter, 1992; Mosses, 1992; Winskel, 1993] discuss operational, denotational and axiomatic approaches and the relations between them. The comparison of rigorous semantics with foundational studies of real numbers is due to D. Scott (quoted in [Meyer and Cosmadakis, 1988]). The concept of full abstraction was introduced in [Milner, 1975].

## 2 A simple imperative language

In this section, we discuss a language consisting of *expressions* and *commands*, also known as ‘statements’ (i.e., imperative statements). The key semantic notion is that of the *state* (of the variables of the computation). In general, the value of an expression depends on the current state; the computing device changes the state when it *executes* a command. The course of the computation is determined by *control structures* such as conditional and iterative commands. Languages that support this style of programming are described as being *imperative*.

The imperative language we consider in this section is ‘simple’ in that there are no jumps, procedures, local declarations, data structures, or modules; but all of these features will be considered in subsequent sections. We will define an abstract syntax, a denotational-semantic interpretation, an operational semantics, and an axiomatic semantics for the language, and verify the correctness of the latter two using the denotational semantics. We will also briefly consider the semantics of a version of the language with *non-deterministic* control structures in Section 2.7; this can be skipped without loss of continuity.

### 2.1 Expressions and commands

#### 2.1.1 Syntax

The types for our initial language fragment are determined by the following BNF-like productions:

$$\begin{aligned}\tau &::= \text{bool} \mid \text{nat} && \text{data types} \\ \theta &::= \text{exp}[\tau] \mid \text{comm} && \text{phrase types};\end{aligned}$$

that is,  $\tau$  and  $\theta$  are meta-variables ranging over the sets  $\{\text{bool}, \text{nat}\}$  and  $\{\text{exp}[\text{bool}], \text{exp}[\text{nat}], \text{comm}\}$ , respectively. The data types represent the sets of Boolean (truth) values and natural numbers, respectively. Additional data types (*int*, *real*, *char*, ...) can be treated analogously. The

phrase types are as follows: expressions for each of the data types, and commands.

The syntax of the language fragment is, for our purposes, sufficiently defined by the inference rules of Table 3. We use  $C$ ,  $B$ , and  $N$  as syntactic meta-variables ranging over the commands, Boolean expressions, and numerical expressions, respectively.

*Bracketing:*

$$\frac{X: \theta}{(X): \theta}$$

*Conditional:*

$$\frac{B: \text{exp}[\text{bool}] \quad X_0: \theta \quad X_1: \theta}{\text{if } B \text{ then } X_0 \text{ else } X_1: \theta}$$

*Zero:*

$$\overline{0: \text{exp}[\text{nat}]}$$

*Successor:*

$$\frac{N: \text{exp}[\text{nat}]}{\text{succ } N: \text{exp}[\text{nat}]}$$

*Truth:*

$$\overline{\text{true}: \text{exp}[\text{bool}]}$$

*Negation:*

$$\frac{B: \text{exp}[\text{bool}]}{\text{not } B: \text{exp}[\text{bool}]}$$

*Conjunction:*

$$\frac{B_0: \text{exp}[\text{bool}] \quad B_1: \text{exp}[\text{bool}]}{B_0 \text{ and } B_1: \text{exp}[\text{bool}]}$$

*Ordering:*

$$\frac{N_0: \text{exp}[\text{nat}] \quad N_1: \text{exp}[\text{nat}]}{N_0 < N_1: \text{exp}[\text{bool}]}$$

*Equality:*

$$\frac{E_0: \text{exp}[\tau] \quad E_1: \text{exp}[\tau]}{E_0 = E_1: \text{exp}[\text{bool}]}$$

*Null:*

$$\overline{\text{skip}: \text{comm}}$$

*Sequencing:*

$$\frac{C_0: \text{comm} \quad C_1: \text{comm}}{C_0 ; C_1: \text{comm}}$$

*Definite Iteration:*

$$\frac{N: \text{exp}[\text{nat}] \quad C: \text{comm}}{\text{for } N \text{ do } C: \text{comm}}$$

**Table 3.** Syntax of the imperative language

In fact, these syntax rules describe only an *abstract* syntax; i.e., a syntax that specifies *what* the sets of phrase structures (derivations) are, but does not determine certain aspects of *how* these phrase structures are represented as linear strings, such as operator precedence and associativity. Technically, the abstract syntax is *syntactically ambiguous*, in that some strings would be assigned more than one phrase structure; for example, there are *two* derivations of **for**  $N$  **do**  $C_0 ; C_1$  from  $N : \text{exp}[\text{nat}]$ ,  $C_0, C_1 : \text{comm}$ .

At first sight this might seem to be a serious problem, leading to ambiguity of the semantic interpretation. But because each well-formed phrase is the conclusion of exactly *one* rule and the valuation functions will be defined compositionally, each *derivation* allowed by the syntax will have an unambiguous meaning, and this is adequate for our purposes. For actual use of (linearized strings of) the language, it would be necessary to define a more refined (or ‘concrete’) syntax that unambiguously specified a unique derivation for every well-formed string. To give concrete examples, we will simply adopt informal conventions or use parentheses as necessary to indicate syntactic structure.

There are two ‘generic’ constructions, bracketing and conditional selection, which are applicable with all of the phrase types. For example, the Conditional rule may be regarded as an abbreviation for the following three rules:

$$\frac{B : \text{exp}[\text{bool}] \quad B_0 : \text{exp}[\text{bool}] \quad B_1 : \text{exp}[\text{bool}]}{\text{if } B \text{ then } B_0 \text{ else } B_1 : \text{exp}[\text{bool}]}$$

$$\frac{B : \text{exp}[\text{bool}] \quad N_0 : \text{exp}[\text{nat}] \quad N_1 : \text{exp}[\text{nat}]}{\text{if } B \text{ then } N_0 \text{ else } N_1 : \text{exp}[\text{nat}]}$$

$$\frac{B : \text{exp}[\text{bool}] \quad C_0 : \text{comm} \quad C_1 : \text{comm}}{\text{if } B \text{ then } C_0 \text{ else } C_1 : \text{comm}}$$

The first two forms are conditional *expressions*, and the third is the more familiar conditional *command*. Note that in each case the two ‘arms’ must have the same phrase type.

The remaining numerical expressions are: a constant and a unary operation on a numerical operand. The remaining Boolean expressions are: a constant, unary and binary operations on Boolean operands, an ordering predicate on numerical operands, and equality predicates for all data types. Additional constants (**false**, 1, 2, ...), predicates ( $\neq, \leq, >, \dots$ ), and operators (**or**,  $\supset, +, \times, \dots$ ) can be treated analogously, and will be used freely whenever convenient.

The remaining commands are: a null command (which has no effect), sequential composition, and a loop. The loop form **for**  $N$  **do**  $C$  is executed by evaluating the numerical expression  $N$  and then executing the body  $C$

that number of times. It is termed a ‘definite’ iteration because the number of repetitions is determined before they begin. The construction

**if  $B$  then  $C$**

will be regarded as an abbreviation for

**if  $B$  then  $C$  else skip**

when  $C$ : **comm**. Initially, we do not provide any state-changing commands; the most important of these, *assignments*, will be considered in the next section.

### 2.1.2 Semantics

Each data-type name  $\tau$  denotes a non-empty set  $\llbracket \tau \rrbracket$  of values possible for some kind of expression:

$$\begin{aligned}\llbracket \mathbf{bool} \rrbracket &= \{true, false\} && \text{truth values} \\ \llbracket \mathbf{nat} \rrbracket &= \{0, 1, 2, \dots\} && \text{natural numbers}\end{aligned}$$

The sets of possible meanings for expression and command phrases are then defined as follows: if  $S$  is a set of states,

$$\begin{aligned}\llbracket \mathbf{exp}[\tau] \rrbracket &= S \rightarrow \llbracket \tau \rrbracket \\ \llbracket \mathbf{comm} \rrbracket &= S \rightarrow S,\end{aligned}$$

where, for any sets  $A$  and  $B$ ,  $A \rightarrow B$  denotes the set of all functions from  $A$  to  $B$ . The meaning of an expression is the function that, for any state, yields the value of the expression relative to that state, and the meaning of a command is some state transformation. The set of states,  $S$ , may be left unspecified for now. These domains reflect our (current) assumptions that an expression has a value at every state, that expression evaluations never change the state, and that command executions always terminate normally.

The use of input-to-output functions as command and expression meanings allows us to abstract away from certain ‘operational’ aspects of program execution. For example, the command **if  $N = N$  then  $C$**  is semantically equivalent to  $C$ , despite the difference in their computational behaviour (in a straightforward implementation). This is only possible because there is no way to observe ‘intermediate’ execution states.

We must now define semantic valuation functions

$$\begin{aligned}\llbracket \cdot \rrbracket_{\mathbf{exp}[\tau]}: \llbracket \mathbf{exp}[\tau] \rrbracket &\rightarrow \llbracket \mathbf{exp}[\tau] \rrbracket \\ \llbracket \cdot \rrbracket_{\mathbf{comm}}: \llbracket \mathbf{comm} \rrbracket &\rightarrow \llbracket \mathbf{comm} \rrbracket\end{aligned}$$

where  $[\theta]$  is, as before, the set of well-formed phrases of type  $\theta$ . A semantic equation for each rule in the syntax is given in Table 4. Further constants,

predicates and operators would be treated analogously. The semantic meta-variable  $s$  ranges over the set  $S$  of states.

$$\begin{aligned}
\llbracket (X) \rrbracket &= \llbracket X \rrbracket \\
\llbracket \text{if } B \text{ then } X_0 \text{ else } X_1 \rrbracket s &= \begin{cases} \llbracket X_0 \rrbracket s, & \text{if } \llbracket B \rrbracket s = \text{true} \\ \llbracket X_1 \rrbracket s, & \text{if } \llbracket B \rrbracket s = \text{false} \end{cases} \\
\llbracket \text{true} \rrbracket s &= \text{true} \\
\llbracket \text{not } B \rrbracket s &= \begin{cases} \text{true}, & \text{if } \llbracket B \rrbracket s = \text{false} \\ \text{false}, & \text{if } \llbracket B \rrbracket s = \text{true} \end{cases} \\
\llbracket B_0 \text{ and } B_1 \rrbracket s &= \begin{cases} \text{true}, & \text{if } \llbracket B_0 \rrbracket s = \text{true} \text{ and } \llbracket B_1 \rrbracket s = \text{true} \\ \text{false}, & \text{otherwise} \end{cases} \\
\llbracket N_0 < N_1 \rrbracket s &= (\llbracket N_0 \rrbracket s < \llbracket N_1 \rrbracket s) \\
\llbracket E_0 = E_1 \rrbracket s &= (\llbracket E_0 \rrbracket s = \llbracket E_1 \rrbracket s) \\
\llbracket 0 \rrbracket s &= 0 \\
\llbracket \text{succ } N \rrbracket s &= (\llbracket N \rrbracket s) + 1 \\
\llbracket \text{skip} \rrbracket s &= s \\
\llbracket C_0 ; C_1 \rrbracket s &= \llbracket C_1 \rrbracket (\llbracket C_0 \rrbracket s) \\
\llbracket \text{for } N \text{ do } C \rrbracket s &= \llbracket C \rrbracket^n(s), \text{ where } n = \llbracket N \rrbracket s
\end{aligned}$$

**Table 4.** Semantic equations for the imperative language

The interpretation of bracketing ensures that brackets only define syntactic structure. The interpretation of the expressions is straightforward: the state argument is inherited by sub-expressions (and will be available to state-dependent forms of expression when these are added), and each feature of the object language is interpreted by the corresponding mathematical concept.

The null command has no effect on the state, and sequencing is interpreted as composition of state-transforming functions. These interpretations can be expressed more directly as follows:

$$\begin{aligned}
\llbracket \text{skip} \rrbracket &= \text{id}_S \\
\llbracket C_0 ; C_1 \rrbracket &= \llbracket C_0 \rrbracket ; \llbracket C_1 \rrbracket
\end{aligned}$$

where  $\text{id}_S(s) = s$  and  $(c_0 ; c_1)(s) = c_1(c_0(s))$ . The use of  $;$  for function composition in ‘diagrammatic order’ is merely a convenience; the semantics would be the same if we were to use  $\cdot$  for function composition in algebraic order and write

$$\llbracket C_0 ; C_1 \rrbracket = \llbracket C_1 \rrbracket \cdot \llbracket C_0 \rrbracket$$

The loop form

$$\text{for } N \text{ do } C$$

is interpreted as an iteration of the state transformation denoted by  $C$ ; for any  $c: S \rightarrow S$ ,  $c^0 = \text{id}_S$  and  $c^{n+1} = c^n ; c$ . Notice that changes to the state in  $C$  do not affect the number of iterations, which is determined by evaluating  $N$  in the initial state.



The command equivalences in Table 5 are easily validated using the semantic valuation.

$$\begin{array}{l}
C ; \mathbf{skip} \equiv_{\text{comm}} C \equiv_{\text{comm}} \mathbf{skip} ; C \\
(C_0 ; C_1) ; C_2 \equiv_{\text{comm}} C_0 ; (C_1 ; C_2) \\
\mathbf{if true then } C_0 \mathbf{ else } C_1 \equiv_{\text{comm}} C_0 \\
\mathbf{if false then } C_0 \mathbf{ else } C_1 \equiv_{\text{comm}} C_1 \\
(\mathbf{if } B \mathbf{ then } C_0 \mathbf{ else } C_1) ; C_2 \equiv_{\text{comm}} \mathbf{if } B \mathbf{ then } (C_0 ; C_2) \mathbf{ else } (C_1 ; C_2) \\
\mathbf{for } 0 \mathbf{ do } C \equiv_{\text{comm}} \mathbf{skip} \\
\mathbf{for succ } N \mathbf{ do } C \equiv_{\text{comm}} (\mathbf{for } N \mathbf{ do } C) ; C \\
\mathbf{for } N \mathbf{ do skip} \equiv_{\text{comm}} \mathbf{skip}
\end{array}$$

Table 5. Command equivalences

For example,

$$\llbracket (C_0 ; C_1) ; C_2 \rrbracket s = \llbracket C_2 \rrbracket (\llbracket C_1 \rrbracket (\llbracket C_0 \rrbracket s)) = \llbracket C_0 ; (C_1 ; C_2) \rrbracket s$$

for every  $s \in S$ , and so  $(C_0 ; C_1) ; C_2 \equiv_{\text{comm}} C_0 ; (C_1 ; C_2)$ . In fact, it can be proved by a structural induction over all forms of command in the language that *every* command is, for now, equivalent to **skip**; however, all of the equivalences listed will remain valid when state-changing assignment commands are added in the next section.

## 2.2 Assignment Commands

The most glaring omissions from the language fragment described in the preceding section are ways to access and change computational states. We therefore assume a set of *variable-identifiers*, ranged over by meta-variable  $\iota$  (the Greek letter *iota*), and a type-assigning function  $\pi_0$  from these to data types; the lexical rules for variable-identifiers need not concern us.

We use  $\text{dom } f$  to denote the domain of any function  $f$ , so that  $\text{dom } \pi_0$  is the set of all variable-identifiers. For every data type  $\tau$ ,

$$[\tau] = \{\iota \in \text{dom } \pi_0 \mid \pi_0(\iota) = \tau\};$$

i.e., the set of  $\tau$ -valued variable-identifiers. (This notation should not be confused with  $[\theta]$  for *phrase* types  $\theta$ .) Then, we augment the syntactic rules for our language as follows:

*Variable-identifier:*

$$\frac{}{\iota: \mathbf{exp}[\tau]} \quad \text{when } \iota \in [\tau]$$

*Assignment:*

$$\frac{E: \mathbf{exp}[\tau]}{\iota := E: \mathbf{comm}} \quad \text{when } \iota \in [\tau]$$

The variable-identifier form of expression and the assignment command allow named components of computational states to be, respectively, accessed and updated.

The following is an example of a code fragment in this language; it has the effect of decrementing (if possible) the value of  $n \in [\mathbf{nat}]$  (using additional variables  $i, j \in [\mathbf{nat}]$ ):

```

i := 0; j := 0;
for n do
  (if i > 0 then
    j := succ j);
  i := succ i;
n := j.

```

We assume that the ‘syntactic scopes’ of all control structures such as **if** and **for** extend as far to the right as possible; for example, assignment  $i := \mathbf{succ} i$  in the above is within the body of the **for** loop, which should in any case be clear from the indentation.

To describe the semantics of assignment commands and variables used as expressions, we define an appropriate set of states as follows:

$$S = \prod_{\iota \in \text{dom } \pi_0} [\pi_0(\iota)],$$

where, if  $A_i$  is a set for every  $i \in I$ ,  $\prod_{i \in I} A_i$  is the set of functions  $f$  from  $I$  to the union of the  $A_i$  such that, for every  $i \in I$ ,  $f(i) \in A_i$ . That is,  $S$  is the set of all functions  $s$  from variable-identifiers to values such that, for every variable-identifier  $\iota \in [\tau]$ ,  $s(\iota) \in [\tau]$ . Then, the semantic equations are as follows:

$$\begin{aligned} \llbracket \iota \rrbracket s &= s(\iota) \\ \llbracket \iota := E \rrbracket s &= (s \mid \iota \mapsto \llbracket E \rrbracket s), \end{aligned}$$

where  $(s \mid \iota \mapsto \llbracket E \rrbracket s)$  is the state  $s'$  such that  $s'(\iota) = \llbracket E \rrbracket s$  and, for all variable-identifiers  $\iota' \neq \iota$ ,  $s'(\iota') = s(\iota')$ . Informally: the value of a variable-identifier at some state is the value in that component of the state, and the effect of executing an assignment in some state is to produce a ‘new’ state

that is identical except that the value in the component for the variable-identifier is replaced by the value of the right-hand side expression at the ‘old’ state.

The following is an example of a command equivalence that can be validated using the valuation for assignment:

$$\iota := (\text{if } B \text{ then } E_0 \text{ else } E_1) \equiv_{\text{comm}} \text{if } B \text{ then } (\iota := E_0) \text{ else } (\iota := E_1)$$

where  $\iota \in [\tau]$ ,  $B: \mathbf{exp}[\mathbf{bool}]$ , and  $E_0, E_1: \mathbf{exp}[\tau]$ , and of course all of the equivalences listed in Table 5 remain valid.

## 2.3 Indefinite iterations

The **for** loop is often inconvenient or inefficient because the number of iterations is determined before the iteration begins. So, let us replace the **for** loop by the following:

*Indefinite Iteration:*

$$\frac{B: \mathbf{exp}[\mathbf{bool}] \quad C: \mathbf{comm}}{\mathbf{while } B \text{ do } C: \mathbf{comm}}$$

Expression  $B$  is evaluated and command  $C$  is executed repeatedly, in alternation, until the value of  $B$  becomes *false*.

Unfortunately, it is possible that the value *never* becomes *false*, and then the execution does not terminate. We therefore re-define the set of command meanings as follows:

$$\llbracket \mathbf{comm} \rrbracket = S \rightarrow S,$$

where, for any sets  $A$  and  $B$ ,  $A \rightarrow B$  is the set of all *partial* functions from  $A$  to  $B$ . The intention is that  $\llbracket C \rrbracket s$  is to be undefined if and only if execution of command  $C$  from initial state  $s$  fails to terminate. No changes to the semantic equations of Sections 2.1 and 2.2 are needed if we view the semi-colon as denoting composition of *partial* functions, and similarly for  $c^n$ .

It is convenient to introduce also the following form of command:

*Divergence:*

$$\overline{\mathbf{diverge: comm}}$$

This command is intended to always fail to terminate, regardless of what the computational state is initially; i.e.,  $\llbracket \mathbf{diverge} \rrbracket s$  is undefined for every  $s \in S$ . The **diverge** command is useless in practice; however, it will be useful in formulating the semantics of the **while** loop.

We want an interpretation of the **while** loop for which the following semantic equivalence holds:

$$\mathbf{while} \ B \ \mathbf{do} \ C \equiv_{\text{comm}} \ \mathbf{if} \ B \ \mathbf{then} \ (C ; \mathbf{while} \ B \ \mathbf{do} \ C). \quad (2.1)$$

This means that we must obtain an interpretation such that

$$\llbracket \mathbf{while} \ B \ \mathbf{do} \ C \rrbracket = \llbracket \mathbf{if} \ B \ \mathbf{then} \ (C ; \mathbf{while} \ B \ \mathbf{do} \ C) \rrbracket.$$

If we let  $c$  stand for the (currently unknown) meaning of **while**  $B$  **do**  $C$  and simplify the right-hand side using the known valuations for **if** and **;**, we get an *equation* of the form

$$c = \dots \llbracket B \rrbracket \dots \llbracket C \rrbracket \dots c \dots$$

such that a solution for  $c$ , the indeterminate of the equation, would be the desired interpretation of the **while** loop. But how can we solve this equation?

Consider the sequence of commands  $C_i$  for  $i \in \omega = \{0, 1, 2, \dots\}$ , defined inductively as follows:

$$\begin{aligned} C_0 &= \mathbf{diverge} \\ C_{i+1} &= \mathbf{if} \ B \ \mathbf{then} \ (C ; C_i) \end{aligned}$$

The right-hand side of the definition of  $C_{i+1}$  is obtained from the right-hand side of (2.1) by replacing **while**  $B$  **do**  $C$  by  $C_i$ . The meaning of each of the  $C_i$  is an *approximation* to the desired meaning of the **while** loop in the sense that, if  $C_i$  terminates for some initial state, then so does **while**  $B$  **do**  $C$ , and the final states will be the same; furthermore, as  $i$  increases, the approximations can only improve in the sense that terminating states (rather than non-termination) can be obtained after longer computations. The  $C_i$  can be thought of as specifying the effect obtained by executing the loop using an implementation that works correctly provided the loop terminates in fewer than  $i$  iterations, but self-destructs otherwise.

These syntactic considerations motivate the following semantic construction. Let  $c_0$  be the partial function on  $S$  that is everywhere undefined; that is,  $\text{graph } c_0 = \emptyset$ , where  $\text{graph } f$  for  $f: A \rightarrow B$  is the set of ordered pairs  $\{(a, b) \mid b = f(a)\}$ . Then, for  $i \in \omega$ , let

$$c_{i+1}(s) = \begin{cases} (\llbracket C \rrbracket ; c_i)(s), & \text{if } \llbracket B \rrbracket s = \text{true} \\ s, & \text{if } \llbracket B \rrbracket s = \text{false}. \end{cases}$$

It is clear that, for every  $i \in \omega$ ,  $c_i = \llbracket C_i \rrbracket$ . Furthermore, it can be shown by mathematical induction that  $\text{graph } c_i \subseteq \text{graph } c_{i+1}$  for every  $i \in \omega$ ; this

allows us to define the partial function  $c_\infty$  on  $S$  whose graph is the *union* of the graphs for all the  $c_i$ :

$$\text{graph } c_\infty = \bigcup_{i \in \omega} \text{graph } c_i.$$

Then we define  $\llbracket \text{while } B \text{ do } C \rrbracket$  to be  $c_\infty$ . Note that this interpretation of the **while** loop is compositional, because  $\llbracket \text{while } B \text{ do } C \rrbracket$  is defined in terms of  $\llbracket B \rrbracket$  and  $\llbracket C \rrbracket$  only, albeit in a more complex manner than usual.

For example, suppose we add a decrementation operator **pred** to the language, and consider the following **while** loop:

$$\text{while } n > 0 \text{ do } n := \text{pred } n; \quad (2.2)$$

then

$$\begin{aligned} C_0 &= \text{diverge} \\ C_1 &= \text{if } n > 0 \text{ then} \\ &\quad n := \text{pred } n; \\ &\quad \text{diverge} \\ C_2 &= \text{if } n > 0 \text{ then} \\ &\quad n := \text{pred } n; \\ &\quad \text{if } n > 0 \text{ then} \\ &\quad \quad n := \text{pred } n; \\ &\quad \text{diverge} \end{aligned}$$

and so on. The partial functions  $c_i = \llbracket C_i \rrbracket$  are as follows:

$$\begin{aligned} c_0(s) &\text{ is undefined for every } s \in S \\ c_1(s) &= \begin{cases} \text{undefined,} & \text{if } \llbracket n \rrbracket s \geq 1 \\ s, & \text{if } \llbracket n \rrbracket s = 0 \end{cases} \\ c_2(s) &= \begin{cases} (\llbracket n := \text{pred } n \rrbracket ; c_1)(s), & \text{if } \llbracket n \rrbracket s > 0 \\ s, & \text{if } \llbracket n \rrbracket s = 0 \end{cases} \\ &= \begin{cases} \text{undefined,} & \text{if } \llbracket n \rrbracket s \geq 2 \\ (s \mid n \mapsto 0), & \text{if } \llbracket n \rrbracket s < 2 \end{cases} \\ &\vdots \\ c_i(s) &= \begin{cases} \text{undefined,} & \text{if } \llbracket n \rrbracket s \geq i \\ (s \mid n \mapsto 0), & \text{if } \llbracket n \rrbracket s < i \end{cases} \\ &\vdots \end{aligned}$$

The meaning of (2.2) is then the function  $c_\infty$  whose graph is the union of the graphs of all the  $c_i$ ; i.e.,  $c_\infty(s) = (s \mid n \mapsto 0)$  for all  $s \in S$ . Hence, (2.2) is semantically equivalent to  $n := 0$ . Note that  $c_\infty$  in this example is actually a *total* function (defined for every possible argument), but every  $c_i$  for  $i \in \omega$  is strictly partial (undefined for some arguments).



The following proposition shows that in general our interpretation of the **while** loop satisfies equivalence (2.1):

**Proposition 2.3.1.** *For any  $B: \text{exp}[\text{bool}]$  and  $C: \text{comm}$ ,*

$$\llbracket \text{while } B \text{ do } C \rrbracket = \llbracket \text{if } B \text{ then } (C ; \text{while } B \text{ do } C) \rrbracket.$$

**Proof.** Consider any  $s \in S$ ; we will show that

$$\llbracket \text{while } B \text{ do } C \rrbracket s = c_\infty(s) \quad (2.3)$$

and

$$\begin{aligned} & \llbracket \text{if } B \text{ then } (C ; \text{while } B \text{ do } C) \rrbracket s \\ &= \begin{cases} (\llbracket C \rrbracket ; c_\infty)(s), & \text{if } \llbracket B \rrbracket s = \text{true} \\ s, & \text{if } \llbracket B \rrbracket s = \text{false} \end{cases} \end{aligned} \quad (2.4)$$

are either both undefined or both defined and equal, where  $c_\infty$  is as defined above. Either

- (2.3) is undefined

$$\begin{aligned} & \iff c_i(s) \text{ is undefined for every } i \in \omega \\ & \iff \llbracket B \rrbracket s = \text{true} \text{ and either } \llbracket C \rrbracket s \text{ is undefined} \\ & \quad \text{or } c_{i-1}(\llbracket C \rrbracket s) \text{ is undefined for every } i > 0 \\ & \iff (2.4) \text{ is undefined,} \end{aligned}$$

or

- (2.3) is defined and equal to  $s' \in S$

$$\begin{aligned} & \iff c_i(s) = s' \text{ for some } i > 0 \\ & \iff (\llbracket B \rrbracket s = \text{false} \text{ and } s = s') \\ & \quad \text{or } (\llbracket B \rrbracket s = \text{true} \text{ and } s' = c_{i-1}(\llbracket C \rrbracket s) \text{ for some } i > 0) \\ & \iff (2.4) \text{ is defined and equal to } s'. \end{aligned}$$

■

Other looping control structures, such as **repeat**  $C$  **until**  $B$ , can be treated analogously; however, ‘jumps’, such as multi-level exits and the **goto**, require more complex techniques, to be discussed in Section 6. The limit construction described in this section will be generalized to treat recursive definitions in Section 4.

**Exercise 2.3.2.** Verify the following command equivalences:

1. **while not true do**  $C \equiv_{\text{comm}}$  **skip**
2. **while true do**  $C \equiv_{\text{comm}}$  **diverge**

3.  $(\text{while } B \text{ do } C) ; \text{if } B \text{ then } C_0 \text{ else } C_1 \equiv_{\text{comm}} (\text{while } B \text{ do } C) ; C_1$

where  $B: \text{exp}[\text{bool}]$  and  $C, C_0, C_1: \text{comm}$ .

**Exercise 2.3.3.** Define the syntax and semantics of the command form

**repeat**  $C$  **until**  $B$

so as to achieve the equivalence

**repeat**  $C$  **until**  $B \equiv_{\text{comm}} C ; \text{if not } B \text{ then repeat } C \text{ until } B,$

and then verify the following equivalence using induction:

**repeat**  $C$  **until**  $B \equiv_{\text{comm}} C ; \text{while not } B \text{ do } C$

## 2.4 Programs

We can introduce a type name **prog** for (complete) programs, and define

$$\llbracket \text{prog} \rrbracket = I \rightarrow O$$

for appropriate sets  $I$  and  $O$  of *inputs* and *outputs*, respectively. To avoid having to deal here with the intricacies of input and output operations, we will regard any command by itself as a complete program, as follows,

*Program:*

$$\frac{C: \text{comm}}{C: \text{prog}}$$

and assume the existence of (partial) functions

$$\begin{aligned} g_0: I &\rightarrow S \\ k_0: S &\rightarrow O \end{aligned}$$

that map inputs to (initial) states, and (final) states to outputs in some standard way; then

$$\llbracket C \rrbracket_{\text{prog}} = g_0 ; \llbracket C \rrbracket_{\text{comm}} ; k_0.$$

For example, suppose that

- input and output consist of single natural numbers;
- the input is made available to the program as the initial value of a variable  $in$ , and all other variables are initialized to standard initial

values  $v_\tau$  for each data type  $\tau$  (such as  $v_{\text{bool}} = \text{false}$  and  $v_{\text{nat}} = 0$ ); and

- the output is produced from the final value of a variable  $out$ .

Then we can define

- $I = O = N$ ;
- $g_0(n) = (s_0 \mid in \mapsto n)$  where  $s_0(\iota) = v_\tau$  when  $\iota \in [\tau]$ ; and
- $k_0(s) = s(out)$ .

As a second example, if programs are executed only to produce an indication that they have terminated, we would take  $I$  and  $O$  to be singleton sets, say  $\{*\}$ , with  $g_0(*) = s_0$  and  $k_0(s) = *$  for every  $s \in S$ . Then, there are exactly two program meanings: the identity function, signifying program termination, and the undefined function, signifying non-termination. Note that the meaning of non-terminating programs ‘approximates’ the meaning of terminating ones, in that the graph of the undefined function is a subset of the graph of the identity function.

**Exercise 2.4.1.** Prove that our semantics is fully abstract.

## 2.5 Operational semantics

In this section, we present an *operational* semantics for our simple imperative language. The style of implementation formalized is not very realistic, but our aim is not to discuss implementation techniques in any detail, but rather to clarify the differences between operational and denotational descriptions, and to demonstrate how the latter can be used to verify the correctness of the former. We simplify the presentation by interpreting only the commands operationally, using the denotational semantics for the expressions.

We will obtain an operational semantics for the commands by defining a binary relation  $\succ$  on ‘machine configurations’ of the form  $(C, s)$ , where command  $C$  is the *program* component, and state  $s \in S$  is the *data* component.<sup>1</sup> The relationship  $(C, s) \succ (C', s')$  should hold just when this change of configuration is a single computational step.

The  $\succ$  relation is defined to be the smallest (with respect to the subset ordering) binary relation on configurations satisfying the axioms and rules of Table 6.

The treatment of command sequencing can be explained intuitively as follows:  $C_0; C_1$  is executed by trying to reduce  $C_0$  (possibly in several steps) to **skip**, which is irreducible, and then reducing  $C_1$ . Table 7 is a (partial)

---

<sup>1</sup>The use of abstract states in the operational configurations simplifies the presentation, but it would be better if configurations were *finitary*; in fact, any program in this language can use only a syntactically-determinable finite set of variables. Also, the data component should really contain *representations* of truth values and numbers, rather than the abstract values themselves, but we will ignore this technicality.

'trace' of a computation according to this operational interpretation;  $s$  is any state.

$$\overline{((C), s) \succ (C, s)} \quad (2.5)$$

$$\overline{(\iota := E, s) \succ (\text{skip}, (s \mid \iota \mapsto \llbracket E \rrbracket s))} \quad (2.6)$$

$$\overline{(\text{while } B \text{ do } C, s) \succ (\text{if } B \text{ then } C ; \text{while } B \text{ do } C, s)} \quad (2.7)$$

$$\frac{\llbracket B \rrbracket s = \text{true}}{(\text{if } B \text{ then } C_1 \text{ else } C_2, s) \succ (C_1, s)} \quad (2.8)$$

$$\frac{\llbracket B \rrbracket s = \text{false}}{(\text{if } B \text{ then } C_1 \text{ else } C_2, s) \succ (C_2, s)} \quad (2.9)$$

$$\overline{(\text{diverge}, s) \succ (\text{diverge}, s)} \quad (2.10)$$

$$\overline{(\text{skip}; C, s) \succ (C, s)} \quad (2.11)$$

$$\frac{(C_0, s) \succ (C'_0, s')}{(C_0 ; C_1, s) \succ (C'_0 ; C_1, s')} \quad (2.12)$$

**Table 6.** Operational semantics of the imperative language

The primary focus of the operational semantics is the notion of a single computational step (in a particular style of implementation); however, it is possible to *derive* the input-to-output behaviour of complete programs. It is easy to verify by structural induction that the  $\succ$  relation is functional; i.e.,  $(C, s) \succ (C', s')$  and  $(C, s) \succ (C'', s'')$  imply  $(C', s') = (C'', s'')$ . This means that we can define a valuation  $\llbracket \cdot \rrbracket$  on commands as follows: for any  $s \in S$ ,

$$\llbracket C \rrbracket s = \begin{cases} s', & \text{if } (C, s) \succ^* (\text{skip}, s') \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

where  $\succ^*$  is the reflexive and transitive closure of  $\succ$ .

We now show that this valuation is equal to the denotational valuation.

**Proposition 2.5.1.** *For all commands  $C$ ,  $\llbracket C \rrbracket = \llbracket C \rrbracket$ .*

$(\text{while } n > 0 \text{ do } n := \text{pred } n, (s \mid n \mapsto 2))$   
 $\succ$  (by (2.7))  
 $(\text{if } n > 0 \text{ then } (n := \text{pred } n; \text{ while } n > 0 \text{ do } n := \text{pred } n), (s \mid n \mapsto 2))$   
 $\succ$  (by (2.8))  
 $((n := \text{pred } n; \text{ while } n > 0 \text{ do } n := \text{pred } n), (s \mid n \mapsto 2))$   
 $\succ$  (by (2.5)),  
 $(n := \text{pred } n; \text{ while } n > 0 \text{ do } n := \text{pred } n, (s \mid n \mapsto 2))$   
 $\succ$  (by (2.6), (2.12))  
 $(\text{skip}; \text{ while } n > 0 \text{ do } n := \text{pred } n, (s \mid n \mapsto 1))$   
 $\succ$  (by (2.11))  
 $(\text{while } n > 0 \text{ do } n := \text{pred } n, (s \mid n \mapsto 1))$   
 $\succ^*$  (as above)  
 $(\text{while } n > 0 \text{ do } n := \text{pred } n, (s \mid n \mapsto 0))$   
 $\succ$  (by (2.7))  
 $(\text{if } n > 0 \text{ then } (n := \text{pred } n; \text{ while } n > 0 \text{ do } n := \text{pred } n), (s \mid n \mapsto 0))$   
 $\succ$  (by (2.9) and the definition of  $\text{if } B \text{ then } C$ )  
 $(\text{skip}, (s \mid n \mapsto 0))$

**Table 7.** Trace of a computation

**Proof.** We first show that  $\text{graph}\{C\} \subseteq \text{graph}[C]$ . It can be directly verified that each of axioms (2.5) to (2.11) of the definition of  $\succ$  in Table 6 has the following property:

$$\llbracket C \rrbracket s = \llbracket C' \rrbracket s' \text{ when } (C, s) \succ (C', s');$$

furthermore,  $\llbracket C_0; C_1 \rrbracket s = \llbracket C'_0; C'_1 \rrbracket s'$  when  $\llbracket C_0 \rrbracket s = \llbracket C'_0 \rrbracket s'$ , so that rule (2.12) preserves this property. But  $(C, s) \succ^* (\text{skip}, s')$  when  $\{C\} s = s'$ , and so we conclude that  $\llbracket C \rrbracket s = \llbracket \text{skip} \rrbracket s' = s'$ .

In the other direction,  $\text{graph}[C] \subseteq \text{graph}\{C\}$  can be proved by structural induction on  $C$ . We discuss the **while** case in detail.

Suppose

$$C = \text{while } B \text{ do } C';$$

the structural-induction hypothesis is that, for all  $s_0, s_1 \in S$ , if  $\llbracket C' \rrbracket s_0 = s_1$  then  $(C', s_0) \succ^* (\text{skip}, s_1)$ . Let commands  $C_i$  for  $i \in \omega$  be the approxima-



tions to **while**  $B$  **do**  $C'$  defined in Section 2.3. We can show by mathematical induction that, for every  $i \in \omega$ ,  $\text{graph}[[C_i]] \subseteq \text{graph}\{\text{while } B \text{ do } C'\}$ .

The basis case is trivial because  $C_0 = \text{diverge}$ . For

$$C_{i+1} = \text{if } B \text{ then } (C' ; C_i),$$

the mathematical-induction hypothesis is that, for all  $s_0, s_1 \in S$ , if  $[[C_i]]s_0 = s_1$  then  $(\text{while } B \text{ do } C', s_0) \succ^* (\text{skip}, s_1)$ . Now, assume that  $[[C_{i+1}]]s = s'$ .

If  $[[B]]s = \text{false}$ , then  $s' = s$  and  $(\text{while } B \text{ do } C', s) \succ^* (\text{skip}, s')$ . On the other hand, if  $[[B]]s = \text{true}$ , let  $s'' = [[C']]s$ , so that  $[[C_i]]s'' = s'$ ; then

$$\begin{aligned} & (\text{while } B \text{ do } C', s) \\ & \quad \succ \quad (\text{if } B \text{ then } (C' ; \text{while } B \text{ do } C'), s) \quad (\text{by (2.7)}) \\ & \quad \succ^* \quad (C' ; \text{while } B \text{ do } C', s) \quad (\text{by } [[B]]s = \text{true} \text{ and (2.8)}) \\ & \quad \succ^* \quad (\text{skip} ; \text{while } B \text{ do } C', s'') \\ & \quad \quad (\text{by } [[C']]s = s'' \text{ and the structural-induction hypothesis}) \\ & \quad \succ \quad (\text{while } B \text{ do } C', s'') \quad (\text{by (2.11)}) \\ & \quad \succ^* \quad (\text{skip}, s') \\ & \quad \quad (\text{by } [[C_i]]s'' = s' \text{ and the mathematical-induction hypothesis}). \end{aligned}$$

In both cases,  $(C, s) \succ^* (\text{skip}, s')$ , and so  $\text{graph}[[C_{i+1}]] \subseteq \text{graph}\{C\}$ . By induction on  $i$ ,  $\text{graph}[[C_i]] \subseteq \text{graph}\{C\}$  for every  $i \in \omega$ , and since

$$\text{graph}[[C]] = \bigcup_{i \in \omega} \text{graph}[[C_i]],$$

we can conclude that  $\text{graph}[[C]] \subseteq \text{graph}\{C\}$ . ■

The proposition shows that the operational description for commands defined in this section is functionally equivalent (but clearly not *identical*) to the denotational description given in preceding sections. This kind of result can be interpreted in different ways.

One view is that a denotational description is conceptually more fundamental than representation-dependent operational interpretations, and so the result shows that the operational description presented is *correct* relative to the more definitive denotational description.

Some authors take the view that an *operational* description is more fundamental because the effects it specifies are observable; in this view, such a proposition should be interpreted as showing that the denotational valuation is ‘computationally adequate’ in the sense that it is consistent with the operational interpretation (at the level of programs).

The difference between these views is a philosophical one, and the reader is invited to choose between them.

**Exercise 2.5.2.** Give an operational semantics for evaluation of the *expressions* and prove the resulting input-to-output behaviour equivalent to that specified by the denotational semantics.

## 2.6 Programming logic

Programming logics are formal systems of axioms and inference rules for reasoning about program meanings, particularly systems that are useful for specifying, developing, transforming, and verifying programs. Sometimes they are termed ‘axiomatic semantics’. In this section we study a system called Floyd–Hoare logic which is widely used for reasoning about simple imperative programs of the kind we have been considering in this section. Most readers will have seen some informal use of the methods in programming courses, and several textbooks discuss the techniques in great detail. Our aim here is not to discuss programming methodology, but rather to clarify the differences between denotational and axiomatic descriptions and to demonstrate how semantics can be used to show the *soundness* of a formal system.

### 2.6.1 Syntax and semantics

First, we need formulas for making assertions about properties of particular states. We add a new phrase type as follows:

$$\theta ::= \dots \mid \text{assert},$$

and let  $\llbracket \text{assert} \rrbracket$  be  $S \rightarrow \llbracket \text{bool} \rrbracket$ , where  $S$  is the set of states. Thus, assertions are similar to Boolean expressions; nevertheless, in general it is appropriate to distinguish them. For example, an assertion language may include notations that are not in the programming language (and may even be uncomputable in principle), such as universal or existential quantification. For now, however, it is convenient to assume that assertions and Boolean expressions have the *same* syntax and semantics.

Second, we need a class of formulas for specifying properties of the meanings of commands, expressions, assertions, and so on. These are termed *specifications* and we use the phrase type **spec**; however, except when indicated otherwise, **spec** will not be in the range of meta-variable  $\theta$ . In particular, we do not want ‘conditional’ specifications of the form **if**  $B$  **then**  $Z_0$  **else**  $Z_1$  for  $Z_0, Z_1$ : **spec** and  $B$ : **exp[bool]**. We define  $\llbracket \text{spec} \rrbracket$  to be  $\{\text{true}, \text{false}\}$ ; note that states play no explicit role here.

The atomic specifications are as follows:

*Hoare triple:*

$$\frac{P: \text{assert} \quad C: \text{comm} \quad Q: \text{assert}}{\{P\}C\{Q\}: \text{spec}}$$

Assertions  $P$  and  $Q$  are termed the *pre-condition* and *post-condition*, respectively, of the Hoare triple  $\{P\}C\{Q\}$ . Informally,  $\{P\}C\{Q\}$  is true just if any terminating execution of  $C$  starting with any state satisfying  $P$  terminates in a final state satisfying  $Q$ . The Hoare triple specifies what command  $C$  is intended to do (when it terminates); i.e., achieve  $Q$  whenever  $P$  is initially true. The Hoare-triple form of specification is termed a *partial-correctness* formula because verification of termination is regarded as a separate obligation that the programmer must fulfil to complete a verification of program correctness.

The semantic equation for the Hoare-triple form of specification is as follows:

$$\begin{aligned} \llbracket \{P\}C\{Q\} \rrbracket \\ = \text{for all } s_0, s_1 \in S, \text{ if } \llbracket P \rrbracket s_0 \text{ and } \llbracket C \rrbracket s_0 = s_1 \text{ then } \llbracket Q \rrbracket s_1 \end{aligned}$$

Although every specification has a truth value that is independent of states, there is here an ‘implicit’ quantification over the set of states  $S$ . Notice that  $C$  trivially satisfies a Hoare-triple specification when execution of  $C$  fails to terminate.

Non-atomic forms of specification formulas can be formed by using conjunction ( $\&$ ) and implication ( $\Rightarrow$ ) as propositional connectives. The language is essentially a quantifier-free fragment of multi-sorted predicate logic, with the Hoare-triple form of specification as an atomic formula; that is,  $\{P\}C\{Q\}$  should be regarded as being a three-place predicate,  $H(P, C, Q)$ .

For any assertion  $P$ , we will use  $\{P\}$  as an abbreviation for

$$\{\text{true}\}\text{skip}\{P\}.$$

$\{P\}$  is termed a *static-assertion* specification because  $P$  must hold at *all* states. This completes our presentation of the syntax and semantics of the *language* of Floyd–Hoare logic.

### 2.6.2 Rules and axioms

A specification  $S$  is said to be *valid* if  $\llbracket S \rrbracket = \text{true}$ . We now outline a *formal system* of axioms and rules for inferring valid specifications, starting with axioms for each of the basic forms of command:

$$\{P\}\text{skip}\{P\}$$

$$\{\text{true}\}\text{diverge}\{\text{false}\}$$

$$\{[P](\iota \mapsto E)\}_\iota := E\{P\}$$

In the axiom scheme for assignments,  $[P](\iota \mapsto E)$  denotes the assertion obtained by substituting  $E$  for all occurrences of variable-identifier  $\iota$  in  $P$ , and similarly for  $[E'](\iota \mapsto E)$  when  $E'$  is an expression. The syntax rule for assignments ensures that  $E \in [\text{exp}[\tau]]$  if  $\iota \in [\tau]$ , and so the type of the phrase is preserved by the substitution and the resulting phrase is syntactically well-formed; however, the substitution should be regarded as 'structural', rather than purely textual; for example,  $[a \times b](b \mapsto c + d)$  should yield  $a \times (c + d)$ , rather than  $a \times c + d$ . As an example of the axiom scheme for assignment, we have

$$\{\text{succ } n > 0\}n := \text{succ } n\{n > 0\}$$

when  $n$  is a numerical variable-identifier.

We now present axioms for each of the composite forms of command:

$$\{P\}C\{Q\} \Rightarrow \{P\}(C)\{Q\}$$

$$\{P\}C_0\{Q\} \& \{Q\}C_1\{R\} \Rightarrow \{P\}C_0; C_1\{R\}$$

$$\begin{aligned} \{P \text{ and } B\}C_0\{Q\} \& \{P \text{ and not } B\}C_1\{Q\} \\ \Rightarrow \{P\}\text{if } B \text{ then } C_0 \text{ else } C_1\{Q\} \end{aligned}$$

$$\{P \text{ and } B\}C\{P\} \Rightarrow \{P\}\text{while } B \text{ do } C\{P \text{ and not } B\}$$

Notice that these axioms are *compositional* in the sense that the consequent of each implication is a Hoare-triple specification of a composite command and the precedent is a conjunction of Hoare-triple specifications for its sub-commands. The assertion  $P$  in the axiom for the **while** loop is known as an *invariant* of the **while** loop, because any execution of the loop body  $C$  preserves it, provided that the loop condition  $B$  is true before the execution.

**Exercise 2.6.1.** Propose a suitable axiom for the **repeat** loop.

To allow reasoning about expressions, we may add an axiom  $\{P\}$  for any 'mathematical fact'  $P$  about any of the data types. For example,  $\text{succ } n > 0$  is a state-independent fact about numerical expressions, and so we can treat the static-assertion specification  $\{\text{succ } n > 0\}$  as an axiom.

The connections between assertional and specificational reasoning are dealt with by the following axioms, which are applicable with any command  $C$ :

*Pre-condition strengthening:*

$$\{P_0 \supset P_1\} \& \{P_1\}C\{Q\} \Rightarrow \{P_0\}C\{Q\}$$

*Post-condition weakening:*

$$\{Q_0 \supset Q_1\} \& \{P\}C\{Q_0\} \Rightarrow \{P\}C\{Q_1\}$$

*Pre-condition disjunction:*

$$\begin{aligned} & \{P_1\}C\{Q\} \& \dots \& \{P_n\}C\{Q\} \\ & \Rightarrow \{P_1 \text{ or } \dots \text{ or } P_n\}C\{Q\} \end{aligned}$$

*Post-condition conjunction:*

$$\begin{aligned} & \{P\}C\{Q_1\} \& \dots \& \{P\}C\{Q_n\} \\ & \Rightarrow \{P\}C\{Q_1 \text{ and } \dots \text{ and } Q_n\} \end{aligned}$$

where  $\supset$  is the implication connective for *assertions*; i.e.,

$$\llbracket P \supset Q \rrbracket s = \text{if } \llbracket P \rrbracket s \text{ then } \llbracket Q \rrbracket s.$$

The last two of these axioms apply when  $n$  is zero, yielding the following special cases for any command  $C$  and assertion  $P$ :

$$\{\text{false}\}C\{P\}$$

$$\{P\}C\{\text{true}\}.$$

Finally, we need axioms or rules for the logical connectives  $\&$  and  $\Rightarrow$ . For reasons that will be discussed in Section 7, it is appropriate to adopt *intuitionistic* (rather than classical) propositional logic. We assume the reader is familiar with one of the formulations of this logic.

We illustrate the resulting system by sketching a proof of the validity of the following specification:

$$\begin{aligned} & \{\text{true}\} \\ & \quad \text{sum} := a; i := 0; \\ & \quad (\text{while not}(i = b) \text{ do} \\ & \quad \quad \text{sum} := \text{succ sum}; i := \text{succ } i) \\ & \quad \{ \text{sum} = a + b \} \end{aligned} \tag{2.13}$$

where  $+$  is defined by the following ‘mathematical facts’:  $a + 0 = a$  and  $a + \text{succ } i = \text{succ}(a + i)$ . The key is to find (by heuristic means) an appropriate invariant assertion for the **while** loop. In this case, we just



note that the Boolean negation of the loop condition is  $i = b$  and generalize post-condition  $sum = a + b$  to  $sum = a + i$ . A formal proof can then be obtained as follows.

1. Verify that the invariant assertion is initially achieved by proving

$$\{\text{true}\} sum := a ; i := 0 \{sum = a + i\},$$

using the axioms for assignments and sequencing, the mathematical fact  $a \doteq a + 0$ , and pre-condition strengthening.

2. Verify that the proposed invariant assertion *is* an invariant of the **while** loop by proving

$$\begin{aligned} &\{sum = a + i \text{ and } \text{not}(i = b)\} \\ &\quad sum := \text{succ } sum ; i := \text{succ } i \\ &\{sum \doteq a + i\}, \end{aligned}$$

using the axioms for assignments and sequencing, the mathematical fact  $\text{succ}(a + i) = a + \text{succ } i$ , and pre-condition strengthening.

3. Use the axioms for sequencing and the **while** loop to prove

$$\begin{aligned} &\{\text{true}\} \\ &\quad sum := a ; i := 0 ; \\ &\quad (\text{while } \text{not}(i = b) \text{ do} \\ &\quad \quad sum := \text{succ } sum ; i := \text{succ } i) \\ &\{sum = a + i \text{ and } i = b\} \end{aligned}$$

4. Verify that the invariant and the Boolean negation of the loop condition together imply the desired post-condition, as follows,

$$sum = a + i \text{ and } i = b \supset sum = a + b,$$

and then use post-condition weakening to derive (2.13).

Because of the importance of the invariant assertion in such proofs, the following notation is often used in programs to document the appropriate invariant:

$$\begin{aligned} &\{\text{invariant: } P\} \\ &\text{while } B \text{ do } C \end{aligned}$$

It may be thought of as an abbreviation of

$$\begin{aligned} &\{P\} \\ &(\text{while } B \text{ do} \\ &\quad \{P \text{ and } B\} C \{P\}) \\ &\{P \text{ and } \text{not } B\} \end{aligned}$$

Notice that for any axiom of the form

$$Z_1 \ \& \ Z_2 \ \& \ \cdots \ \& \ Z_n \ \Rightarrow \ Z$$

we can *derive* the inference rule

$$\frac{Z_1 \quad Z_2 \quad \cdots \quad Z_n}{Z}$$

using the usual rules of  $\&$ -Introduction ( $\&$ -I) and  $\Rightarrow$ -Elimination ( $\Rightarrow$ -E), as follows:

$$\frac{\frac{Z_1 \quad Z_2 \quad \cdots \quad Z_n}{Z_1 \ \& \ Z_2 \ \& \ \cdots \ \& \ Z_n} \ (\&\text{-I}) \quad \frac{Z_1 \ \& \ Z_2 \ \& \ \cdots \ \& \ Z_n \Rightarrow Z}{Z} \ (\Rightarrow\text{-E})}{Z}$$

Since *all* of the axioms of the system have this form for *atomic* formulas  $Z_i$  and  $Z$ , it is possible to re-formulate the entire system without the specification connectives  $\&$  and  $\Rightarrow$ ; in fact, Floyd-Hoare logic is conventionally presented in this connective-free style. The format adopted here is convenient for extending the system to handle procedures.

### 2.6.3 Soundness

Our aim in this section is to show that the formal system presented above is *sound*; i.e., that every formula derivable using it is in fact valid according to the semantic interpretation of the formulas. We know that each of the inference rules of propositional intuitionistic logic *preserves* validity, and so we need only prove that each axiom is valid. For most of the axioms, it is straightforward to show validity using the semantic equations. We will discuss only the assignment axiom and the axiom for the **while** loop in detail.

The validity of the assignment axiom for our simple language follows immediately from the following result.

**Proposition 2.6.2.** *If  $E_0 : \text{exp}[\tau_0]$ ,  $E_1 : \text{exp}[\tau_1]$ , and  $\iota \in [\tau_1]$ , then, for all  $s \in S$ ,*

$$\llbracket [E_0](\iota \mapsto E_1) \rrbracket(s) = \llbracket E_0 \rrbracket(s \mid \iota \mapsto \llbracket E_1 \rrbracket s).$$

**Proof.** By structural induction over  $E_0$ . We discuss a few representative cases in detail. Throughout,  $s'$  will denote  $(s \mid \iota \mapsto \llbracket E_1 \rrbracket s)$ .

Case  $E_0 = 0$ :  $\llbracket 0 \rrbracket s = 0 = \llbracket 0 \rrbracket s'$ .

Case  $E_0 = \text{succ } E$ :

$$\llbracket [\text{succ } E](\iota \mapsto E_1) \rrbracket s$$

$$\begin{aligned}
&= \llbracket \mathbf{succ} [E](\iota \mapsto E_1) \rrbracket s \\
&= \llbracket [E](\iota \mapsto E_1) \rrbracket s + 1 \\
&= \llbracket E \rrbracket s' + 1 \text{ (using the induction hypothesis)} \\
&= \llbracket \mathbf{succ} E \rrbracket s'.
\end{aligned}$$

Case  $E_0 = \iota$ :  $\llbracket [\iota](\iota \mapsto E_1) \rrbracket s = \llbracket E_1 \rrbracket s = s'(\iota) = \llbracket \iota \rrbracket s'$ .

Case  $E_0 = \iota_0 \neq \iota$ :  $\llbracket [\iota_0](\iota \mapsto E_1) \rrbracket s = \llbracket \iota_0 \rrbracket s = s(\iota_0) = s'(\iota_0)$  (because  $\iota_0 \neq \iota$ )  $= \llbracket \iota_0 \rrbracket s'$ . ■

To validate the axiom  $\{[P](\iota \mapsto E)\}_\iota := E\{P\}$ , consider states  $s_0$  and  $s_1$  such that  $\llbracket [P](\iota \mapsto E) \rrbracket s_0 = \text{true}$  and  $\llbracket \iota := E \rrbracket s_0 = s_1$ ; then,

$$s_1 = (s_0 \mid \iota \mapsto \llbracket E \rrbracket s_0)$$

by the semantic equation for assignments, and

$$\llbracket [P](\iota \mapsto E) \rrbracket s_0 = \llbracket P \rrbracket (s_0 \mid \iota \mapsto \llbracket E \rrbracket s_0)$$

by the proposition. We conclude that  $\llbracket P \rrbracket s_1 = \text{true}$ , as desired.

Note the importance of the fact that the state at which an expression is evaluated is also the state used for evaluating all of its sub-expressions. Also, it is critical that updating the value of a variable-identifier has no effect on the value of any *other* identifier. Unfortunately, most programming languages do not have these properties.

The validity of the **while** axiom follows immediately from the following result.

**Proposition 2.6.3.** *If  $\llbracket \{P \text{ and } B\}C\{P\} \rrbracket = \text{true}$ , then, for every  $i \in \omega$ ,  $\llbracket \{P\}C_i\{P \text{ and not } B\} \rrbracket = \text{true}$ ,*

where  $C_0, C_1, \dots$  are the commands whose meanings approximate that of the loop **while**  $B$  **do**  $C$ , as defined in Section 2.3.

**Proof.** By mathematical induction on  $i$ . The case  $i = 0$  is trivial because  $C_0 = \mathbf{diverge}$ . Now, assume the result for  $C_i$  and consider any  $s, s' \in S$  such that  $\llbracket P \rrbracket s$  and  $\llbracket C_{i+1} \rrbracket s = c_{i+1}(s) = s'$ . If  $\llbracket B \rrbracket s = \text{false}$  then  $s = s'$  and hence  $\llbracket P \text{ and not } B \rrbracket s' = \text{true}$ . If  $\llbracket B \rrbracket s = \text{true}$ , let  $s''$  be  $\llbracket C \rrbracket s$ ; then  $\llbracket P \rrbracket s'' = \text{true}$  from the assumption, and hence  $\llbracket P \text{ and not } B \rrbracket s' = \text{true}$  by the induction hypothesis. In both cases we have  $\llbracket P \text{ and not } B \rrbracket s' = \text{true}$ . ■

To validate the **while** axiom, assume  $\llbracket \{P \text{ and } B\}C\{P\} \rrbracket = \text{true}$  and consider states  $s_0$  and  $s_1$  such that  $\llbracket P \rrbracket s_0 = \text{true}$  and  $\llbracket \mathbf{while } B \text{ do } C \rrbracket s_0 = s_1$ ; then  $\llbracket C_i \rrbracket s_0 = s_1$  for some  $i \in \omega$  and so, by the proposition,

$$\llbracket P \text{ and not } B \rrbracket s_1 = \text{true},$$

as desired.

**Exercise 2.6.4.** Validate the axiom for the **repeat** loop from Exercise 2.6.1.

We conclude that the formal system presented is *sound* with respect to the denotational interpretation presented. Some authors regard an axiomatic description as definitive, and would interpret this result as showing that the denotational semantics provides a concrete model ‘consistent with’ the axioms. However, it must be kept in mind that there are ‘non-standard’ models of the axioms which may be undesirable or even unimplementable; for example, all of the axioms would be validated if, for every command  $C$ ,  $\llbracket C \rrbracket s$  were undefined for all  $s \in S$ !

Ideally, the formal system would be *complete* relative to the *intended* interpretation, in the sense that every valid formula would be derivable. Completeness considerations are outside the scope of this chapter; see the references in the Bibliographic notes.

## 2.7 Non-determinism

The semantic description of a language need not *fully* specify the result of every computation. It is possible and desirable sometimes to allow implementations some flexibility. Language features whose computational results are not fully determined are described as being *non-deterministic*.

There are several motivations for non-determinacy. One is to allow implementors to take advantage of special properties of particular hardware. For example, floating-point arithmetic cannot realistically be fully-determined by a language specification because of hardware differences. Another motivation is to allow programmers to make explicit in their programs that there may be several equally acceptable computational paths to a correct result or even several correct results. An important example of non-determinacy arises with concurrent processes when the relative speeds of the processors are not determined.

The following control structures are described in [Dijkstra, 1976]:

*Non-deterministic selection:*

$$\frac{B_j: \text{exp}[\text{bool}] \quad C_j: \text{comm, for } j = 1, 2, \dots, n}{\text{if } B_1 \rightarrow C_1 \mid \dots \mid B_n \rightarrow C_n \text{ fi: comm}}$$

*Non-deterministic iteration:*

$$\frac{B_j: \text{exp}[\text{bool}] \quad C_j: \text{comm, for } j = 1, 2, \dots, n}{\text{do } B_1 \rightarrow C_1 \mid \dots \mid B_n \rightarrow C_n \text{ od: comm}}$$

The Boolean expressions are termed ‘guards’. The **if** form is executed by selecting for execution any of the sub-commands  $C_j$  whose corresponding guard  $B_j$  is *true*. If none are *true*, the execution fails to terminate, and if

more than one is *true*, the implementation can arbitrarily select any one of them for execution. For example, the following assigns *true* or *false* to variable *b* if *m* is greater than, or less than *n*, respectively, but can do *either* if  $m = n$ :

```

if   $m \leq n \rightarrow b := \text{true}$ 
    |   $m \geq n \rightarrow b := \text{false}$ 
fi

```

The **do** form is executed by repeatedly selecting for execution any of the  $C_j$  whose guard is *true*, but terminating the repetition when all the guards are *false*.

Semantically, we temporarily re-define the domain of command meanings as follows:

$$\llbracket \text{comm} \rrbracket = \mathcal{P}(S \times S)$$

where  $\mathcal{P}(\cdot)$  is the usual power-set constructor; that is, a command meaning is a binary relation on states. This approach is adequate for treating partial-correctness axioms, but fails to distinguish between a command that, for some initial state, sometimes fails to terminate, and another that has the same set of possible final states but always terminates. More sophisticated approaches to the semantics of non-determinism are beyond the scope of this chapter.

We begin by re-defining the valuations for the null command and for sequential composition of commands:

$$\begin{aligned} \llbracket \text{skip} \rrbracket &= \{(s_0, s_1) \mid s_1 = s_0\} \\ \llbracket C_0 ; C_1 \rrbracket &= \llbracket C_0 \rrbracket ; \llbracket C_1 \rrbracket \end{aligned}$$

where  $c_0 ; c_1 = \{(s_0, s_1) \mid \text{for some } s \in S, (s_0, s) \in c_0 \text{ and } (s, s_1) \in c_1\}$ .

The non-deterministic control structures may then be interpreted as follows:

$$\begin{aligned} \llbracket \text{if } B_1 \rightarrow C_1 \mid \cdots \mid B_n \rightarrow C_n \text{ fi} \rrbracket \\ = \{(s_0, s_1) \mid \text{for some } 1 \leq j \leq n, \llbracket B_j \rrbracket s_0 = \text{true} \text{ and } (s_0, s_1) \in \llbracket C_j \rrbracket\} \\ \llbracket \text{do } B_1 \rightarrow C_1 \mid \cdots \mid B_n \rightarrow C_n \text{ od} \rrbracket = \bigcup_{i \in \omega} c_i \end{aligned}$$

where  $c_0 = \emptyset$  and

$$\begin{aligned} c_{i+1} = \{(s_0, s_1) \mid & (\text{for some } 1 \leq j \leq n, \\ & \llbracket B_j \rrbracket s_0 = \text{true} \text{ and } (s_0, s_1) \in (\llbracket C_j \rrbracket ; c_i)) \\ \text{or, for all } 1 \leq j \leq n, \\ & \llbracket B_j \rrbracket s_0 = \text{false} \text{ and } s_1 = s_0\} \end{aligned}$$

and the valuation for the Hoare-triple specification becomes

$$\llbracket \{P\}C\{Q\} \rrbracket = \text{for all } s_0, s_1 \in S, \text{ if } \llbracket P \rrbracket s_0 \text{ and } (s_0, s_1) \in \llbracket C \rrbracket \text{ then } \llbracket Q \rrbracket s_1.$$

Note that the programmer must ensure that *every* possible result is correct.



## 2.8 Bibliographic notes

The language discussed in this section (except for the non-deterministic control structures) is generally known in the literature as the language of ‘**while** programs’. The operational semantics is based on [Milner, 1976; Plotkin, 1981]. The programming logic is based on [Floyd, 1967; Hoare, 1969], but has been re-formulated here as a multi-sorted quantifier-free first-order theory. The role of Floyd–Hoare logic in programming is discussed in [Alagić and Arbib, 1978; Reynolds, 1981a; Backhouse, 1986], among many others. The first demonstration of the soundness of the system appears to have been [Lauer, 1971]. Completeness issues are discussed in [de Bakker and Meertens, 1975; Cook, 1978; Wand, 1978; Apt, 1981; Bergstra *et al.*, 1982; Bergstra and Tucker, 1982; Goldblatt, 1982; Clarke, 1984; Leivant, 1985a; Pasztor, 1986; Leivant and Fernando, 1987; Abramsky, 1987; Robinson, 1987; Hortalá-González *et al.*, 1988; Tucker and Zucker, 1988]. A survey of programming logics may be found in [Cousot, 1990].

More sophisticated approaches to the semantics of non-determinism are presented in [Plotkin, 1976; Lehmann, 1976; Smyth, 1978; Hennessy and Plotkin, 1979; Plotkin, 1982; Back, 1983; Abramsky, 1983b; Smyth, 1983; Abramsky, 1983a; Winskel, 1983; Connelly, 1990].

Many workers in the area of programming methodology have adopted an approach to command semantics known as ‘predicate transformers’ [Dijkstra, 1976]. In their view, any explicit mention of computational states is undesirably ‘operational’, and they prefer to treat the denotation of a command as a function that maps any desired post-condition to the weakest pre-condition that ensures that execution of the command terminates in a state satisfying that post-condition. [Plotkin, 1979] shows that this approach to command meanings is in fact isomorphic to the state-transformation approach, and so there is no substantive difference between them.

## 3 A simple applicative language

In the programming language discussed in Section 2, all identifiers are accessible globally and there are no procedures. In this section, we consider a language in which

- there are *local* definitions of identifiers, rather than assignment commands that change the values of global variables;
- the programmer can use and define *functions*; and
- the main ‘control structure’ is function application, rather than sequencing and iteration.

Languages that emphasize ‘mathematical’ features (such as definitions, functions, and recursion) are termed *applicative* or *declarative* or *functional*, in contrast to *imperative* languages, which emphasize ‘algorithmic’ features (such as assignments, loops, and jumps). The language to be described in this section is termed ‘simple’ because it does not have recursive definitions; recursion will be considered separately in Section 4.

The key semantic concept to be introduced is that of the *environment*, an assignment of phrase meanings to the identifiers that are used in a

phrase without being bound there. ‘Local’ environments are created by identifier-binding constructions such as definitions. We will discuss the abstract syntax and denotational semantics of the language and a programming logic; an operational semantics for the language extended by recursive definitions will be given in Section 4.

### 3.1 Definitions and function applications

We begin by describing a language fragment consisting of expressions similar to those of Section 2, function applications, and local-definition blocks. The local-definition block has the form

$$\text{let } \iota \text{ be } P \text{ in } Q$$

Intuitively, this allows  $\iota$  to be used in  $Q$  as a symbolic name for the meaning of phrase  $P$ . This might be a *re-definition* of identifier  $\iota$ ; but this is not the same as an *assignment* to a variable, because the scope of the definition is at most the sub-phrase  $Q$ . In implementation terms, the ‘old’ value of a variable is overwritten by an assignment, but the ‘old’ meaning of a re-defined identifier must be retained.

For example, the value of

$$\begin{aligned} &\text{let } n \text{ be } 2 \text{ in} \\ &\quad (\text{let } n \text{ be } n + 1 \text{ in } n) + n \end{aligned} \tag{3.1}$$

is 5. The inner definition of  $n$  has no effect on the occurrence of  $n$  outside the parentheses. We are assuming that the syntactic scope of **let** extends as far to the right as possible, so that that occurrence of  $n$  is bound by the *outer* **let**; furthermore, the occurrence of  $n$  in the phrase  $n + 1$  is also bound by the outer **let**.

The types for our language are defined as follows:

$$\begin{aligned} \tau &::= \text{bool} \mid \text{nat} && \text{data types} \\ \theta &::= \text{val}[\tau] \mid \theta \rightarrow \theta' \mid (\theta) && \text{phrase types.} \end{aligned}$$

Phrases having types of the form  $\text{val}[\tau]$  will be termed *value phrases*. Syntactically, they will be similar to the expressions considered previously. We have used a different phrase type in order to emphasize the important *semantic* difference between the expressions of Section 2 and value phrases here: the latter are ‘pure’ in that their values are *state-independent*, that is to say, not subject to change by assignment commands.

A type of the form  $\theta \rightarrow \theta'$  is a *functional* type;  $\theta$  is the type of arguments to the functions, and  $\theta'$  is the type of the results. A functional type  $\theta \rightarrow \theta'$

is termed *higher-order* if  $\theta$  is a functional type or  $\theta'$  is higher-order. For example,

$$(\text{val}[\text{nat}] \rightarrow \text{val}[\text{bool}]) \rightarrow \text{val}[\text{bool}]$$

is a higher-order type. The function-space type constructor can be used any (finite) number of times to construct a type, and so the language has an *infinite* number of types. We will discuss multi-argument functions later.

The syntax of our language is defined by the inference rules of Table 8.

Most of the rules are similar to the syntax rules for expressions in Section 2.1, except that types of the form  $\text{exp}[\tau]$  have been replaced by types of the form  $\text{val}[\tau]$ . Whenever convenient, we will use additional constants ( $1, \text{false}, \dots$ ) and operators ( $+, \times, \dots$ ) in examples.

The rules for applications and local definitions are new *generic* rules, applicable with all phrase types  $\theta$  and  $\theta'$ . The syntax for applications allows *any* phrase of a functional type to be applied to *any* actual-parameter phrase whose type is the argument type of the functional type. In many languages, the function part of an application must be an identifier and the actual parameter must be parenthesized; but there are no semantic reasons for these syntactic conventions.

Before discussing the rule for local definitions, we must explain how the syntax rules allow for *uses* of identifiers. In the simple imperative language of Section 2, all identifiers were *variable* identifiers and had *global* scope. In the language being considered here, identifiers can be defined to denote meanings of *arbitrary* phrase types and have *local* scopes. Therefore, our syntax must allow for assignments of arbitrary phrase types to (at least) the identifiers that can appear in phrases without being bound there.

This is elegantly achieved by regarding the syntax rules as being *natural-deduction* rules with sequents of the form  $\pi \vdash X : \theta$ , where  $\pi$  is a *phrase-type assignment*, a finite set of assumptions of the form  $\iota : \theta$  for distinct identifiers  $\iota$ . The sequent  $\pi \vdash X : \theta$  asserts that  $X$  is a well-formed phrase of type  $\theta$  *provided* that the identifiers that are used in  $X$  without being bound there have the types assigned to them by  $\pi$ . The natural-deduction framework implicitly provides the following syntax axiom, which allows identifiers to be *used* in well-formed phrases: for any phrase-type assignment  $\pi$ ,  $\pi \vdash \iota : \theta$  when  $\iota : \theta$  is in  $\pi$ . For example,

$$\{\dots, n : \text{val}[\text{nat}], \dots\} \vdash n : \text{val}[\text{nat}],$$

but not

$$\{\dots, n : \text{val}[\text{nat}], \dots\} \vdash n : \text{val}[\text{bool}]$$

or

$$\{n : \text{val}[\text{nat}]\} \vdash m : \text{val}[\text{nat}].$$

*Bracketing:*

$$\frac{X: \theta}{(X): \theta}$$

*Conditional:*

$$\frac{B: \text{val}[\text{bool}] \quad X_0: \theta \quad X_1: \theta}{\text{if } B \text{ then } X_0 \text{ else } X_1: \theta}$$

*Zero:*

$$\overline{0: \text{val}[\text{nat}]}$$

*Successor:*

$$\frac{N: \text{val}[\text{nat}]}{\text{succ } N: \text{val}[\text{nat}]}$$

*Truth:*

$$\overline{\text{true}: \text{val}[\text{bool}]}$$

*Negation:*

$$\frac{B: \text{val}[\text{bool}]}{\text{not } B: \text{val}[\text{bool}]}$$

*Conjunction:*

$$\frac{B_0: \text{val}[\text{bool}] \quad B_1: \text{val}[\text{bool}]}{B_0 \text{ and } B_1: \text{val}[\text{bool}]}$$

*Ordering:*

$$\frac{N_0: \text{val}[\text{nat}] \quad N_1: \text{val}[\text{nat}]}{N_0 < N_1: \text{val}[\text{bool}]}$$

*Equality:*

$$\frac{E_0: \text{val}[\tau] \quad E_1: \text{val}[\tau]}{E_0 = E_1: \text{val}[\text{bool}]}$$

*Application:*

$$\frac{P: \theta \rightarrow \theta' \quad Q: \theta}{P Q: \theta'}$$

*Local definition:*

$$\frac{\begin{array}{c} [\iota: \theta] \\ \vdots \\ P: \theta \quad Q: \theta' \end{array}}{\text{let } \iota \text{ be } P \text{ in } Q: \theta'}$$

**Table 8.** Syntax of the applicative language

A phrase-type assignment  $\pi$  can be thought of as the mathematical counterpart of some state of the ‘symbol table’ in a compiler.

To allow for *pre-defined* identifiers, we assume an *initial* phrase-type assignment  $\pi_0$ , which specifies the phrase types of these identifiers, and

deem  $E$  to be well-formed as a *program* (of type  $\tau$ ) if  $\pi \vdash E: \mathbf{val}[\tau]$  for some finite subset  $\pi$  of  $\pi_0$ .

All of the syntax rules in Table 8 should now be interpreted as natural deduction rules. The Negation rule, for example, specifies that, for any phrase-type assignment  $\pi$ , if  $\pi \vdash B: \mathbf{val}[\mathbf{bool}]$ , then  $\pi \vdash \mathbf{not } B: \mathbf{val}[\mathbf{bool}]$ . For example,

$$\{\dots, b: \mathbf{val}[\mathbf{bool}], \dots\} \vdash \mathbf{not } b: \mathbf{val}[\mathbf{bool}]$$

is derivable using this rule.

The Local-definition rule in Table 8 can now be explained. It will here be convenient to regard a type assignment  $\pi$  as being a *function* from a finite set  $\text{dom } \pi$  of identifiers to phrase types, so that, for all  $\iota \in \text{dom } \pi$ ,  $\pi(\iota)$  is the assumed type of  $\iota$ . The Local-definition rule should be interpreted as stating that, for any  $\pi$ , if  $\pi \vdash P: \theta$  and  $(\pi \mid \iota \mapsto \theta) \vdash Q: \theta'$  then

$$\pi \vdash \mathbf{let } \iota \mathbf{ be } P \mathbf{ in } Q: \theta',$$

where  $(\pi \mid \iota \mapsto \theta)$  is the phrase-type assignment  $\pi'$  such that  $\text{dom } \pi' = \text{dom } \pi \cup \{\iota\}$ ,  $\pi'(\iota) = \theta$ , and  $\pi'(\iota') = \pi(\iota')$  for all  $\iota' \in \text{dom } \pi$  distinct from  $\iota$ . This rule allows an identifier to be re-defined, but gives priority to the most local definition of that identifier, as is conventional in programming languages. Notice also that the rule makes it clear that the scope of the definition includes  $Q$  but not  $P$  or any other phrase.

As an example of how this logical approach to syntax works, we will develop the syntactic derivation of value phrase (3.1). For this example, we assume additional obvious axioms for the numerical constants 1 and 2, and an additional rule as follows:

*Addition:*

$$\frac{N_0: \mathbf{val}[\mathbf{nat}] \quad N_1: \mathbf{val}[\mathbf{nat}]}{N_0 + N_1: \mathbf{val}[\mathbf{nat}]}$$

We start with a derivation of the ‘inner’ **let** using the Addition rule (+) and the Local-definition rule (**let**):

$$\frac{\frac{n: \mathbf{val}[\mathbf{nat}] \quad \overline{1: \mathbf{val}[\mathbf{nat}]}}{n + 1: \mathbf{val}[\mathbf{nat}]} (+) \quad [n: \mathbf{val}[\mathbf{nat}]]}{\mathbf{let } n \mathbf{ be } n + 1 \mathbf{ in } n: \mathbf{val}[\mathbf{nat}]} (\mathbf{let})$$

Notice that the assumption  $n: \mathbf{val}[\mathbf{nat}]$  in the second premiss has been cancelled by using the Local-definition rule (**let**), but the same assumption in the first premiss is still uncanceled. The next step uses the Addition rule again as follows:



$$\frac{\frac{n: \text{val}[\text{nat}] \quad \overline{1: \text{val}[\text{nat}]}}{n+1: \text{val}[\text{nat}]} \quad [n: \text{val}[\text{nat}]]}{\frac{\text{let } n \text{ be } n+1 \text{ in } n: \text{val}[\text{nat}] \quad n: \text{val}[\text{nat}]}{(\text{let } n \text{ be } n+1 \text{ in } n) + n: \text{val}[\text{nat}]}} (+)$$

Another use of the Local-Definition rule then allows the cancellation of two occurrences of the assumption  $n: \text{val}[\text{nat}]$  and gives us the complete desired derivation of (3.1), with no uncanceled assumptions:

$$\frac{\frac{\frac{[n: \text{val}[\text{nat}]] \quad \overline{1: \text{val}[\text{nat}]}}{n+1: \text{val}[\text{nat}]} \quad [n: \text{val}[\text{nat}]]}{\text{let } n \text{ be } n+1 \text{ in } n: \text{val}[\text{nat}] \quad [n: \text{val}[\text{nat}]]} \quad \frac{2: \text{val}[\text{nat}]}{(\text{let } n \text{ be } n+1 \text{ in } n) + n: \text{val}[\text{nat}]}}{\text{let } n \text{ be } 2 \text{ in } (\text{let } n \text{ be } n+1 \text{ in } n) + n: \text{val}[\text{nat}]} (\text{let})$$

We finally can turn to the semantics of the language. The sets of possible meanings for the phrase types are defined as follows:

$$\begin{aligned} \llbracket \text{val}[\tau] \rrbracket &= \llbracket \tau \rrbracket \\ \llbracket \theta \rightarrow \theta' \rrbracket &= \llbracket \theta \rrbracket \rightarrow \llbracket \theta' \rrbracket \\ \llbracket (\theta) \rrbracket &= \llbracket \theta \rrbracket \end{aligned}$$

where, as before,  $\llbracket \tau \rrbracket$  is the set of values for data type  $\tau$ . On the right-hand side,  $A \rightarrow B$  denotes the set of all (total) functions from  $A$  to  $B$ . Note that we must use an inductive definition because there is an infinite number of functional phrase types.

The reader may by now be wondering how *identifiers* are going to be treated. Consider a phrase  $P$ ; the *type* of  $P$  can depend in general on the types of the identifiers used within it, and these are determined by a type assignment  $\pi$  such that  $\pi \vdash P: \theta$ . Similarly, the *meaning* of  $P$  can depend on the meanings of the identifiers used within it; these will be determined by a ‘phrase-meaning assignment’ or *environment*  $u$  which is  $\pi$ -compatible in the sense that, for all  $\iota \in \text{dom } \pi$ ,  $u(\iota) \in \llbracket \pi(\iota) \rrbracket$ . For any phrase-type assignment  $\pi$ , we can define the set of  $\pi$ -compatible environments,  $\llbracket \pi \rrbracket$ , as follows:

$$\llbracket \pi \rrbracket = \prod_{\iota \in \text{dom } \pi} \llbracket \pi(\iota) \rrbracket;$$

i.e., the set of all functions  $u$  with  $\text{dom } u = \text{dom } \pi$  such that, for all  $\iota \in \text{dom } \pi$ ,  $u(\iota) \in \llbracket \pi(\iota) \rrbracket$ .

Let  $[\theta]_\pi$  be the set of all phrases  $P$  such that  $\pi \vdash P: \theta$ . We can now define valuations

$$\llbracket \cdot \rrbracket_{\pi\theta}: [\theta]_\pi \rightarrow (\llbracket \pi \rrbracket \rightarrow \llbracket \theta \rrbracket)$$

mapping every phrase  $P$  such that  $\pi \vdash P: \theta$  into a *function* from  $\llbracket \pi \rrbracket$  to  $\llbracket \theta \rrbracket$ . The  $\pi$  and  $\theta$  subscripts (or just the  $\pi$ ) on phrase-valuation brackets will often be omitted.

For an identifier  $\iota \in \text{dom } \pi$  with  $\pi(\iota) = \theta$ , the semantic equation is

$$\llbracket \iota \rrbracket_{\pi\theta}(u) = u(\iota),$$

for all  $u \in \llbracket \pi \rrbracket$ . To allow for the pre-defined identifiers, we assume a  $\pi_0$ -compatible initial environment  $u_0 \in \llbracket \pi_0 \rrbracket$  that determines their meanings, and interpret *programs* relative to the appropriate restrictions of  $u_0$ .

For most value phrases  $V$ ,  $\llbracket V \rrbracket_{\pi \text{ val}[\tau]}$  can be defined in the same way that the corresponding expression was defined in Section 2.1, but with environment arguments replacing state arguments throughout, as in

$$\llbracket B_0 \text{ and } B_1 \rrbracket u = \begin{cases} \text{true}, & \text{if } \llbracket B_0 \rrbracket u = \text{true} \text{ and } \llbracket B_1 \rrbracket u = \text{true} \\ \text{false}, & \text{otherwise.} \end{cases}$$

The given environment is passed down to each sub-phrase, even if modified environments are created locally within them to handle identifier-binding constructions. Similarly, the conditional form can be interpreted as follows:

$$\llbracket \text{if } B \text{ then } X_0 \text{ else } X_1 \rrbracket_{\theta}(u) = \begin{cases} \llbracket X_0 \rrbracket_{\theta}(u), & \text{if } \llbracket B \rrbracket u = \text{true} \\ \llbracket X_1 \rrbracket_{\theta}(u), & \text{if } \llbracket B \rrbracket u = \text{false.} \end{cases}$$

The semantic equation for function applications is as follows:

$$\llbracket PQ \rrbracket u = (\llbracket P \rrbracket u)(\llbracket Q \rrbracket u);$$

the syntax and the interpretation of the phrase types ensure that  $\llbracket P \rrbracket u$  is a mathematical function and that  $\llbracket Q \rrbracket u$  is an element of its domain to which the function can be *applied*. We can reduce the number of parentheses needed by assuming that function application in the meta-language associates to the left:

$$\llbracket PQ \rrbracket u = \llbracket P \rrbracket u(\llbracket Q \rrbracket u).$$

Finally, the semantic equation for the local-definition block is

$$\llbracket \text{let } \iota \text{ be } P \text{ in } Q \rrbracket u = \llbracket Q \rrbracket (u \mid \iota \mapsto \llbracket P \rrbracket u),$$

where the notation  $(u \mid \iota \mapsto m)$  is analogous to  $(\pi \mid \iota \mapsto \theta)$ ; i.e., the body of the block,  $Q$ , is interpreted relative to a local environment that is like  $u$ , except that  $\iota$  is bound to the meaning of  $P$  in the original environment. The difference between this and the valuation for assignment commands in Section 2.2 should be particularly noted:

$$\llbracket \iota := E \rrbracket s = (s \mid \iota \mapsto \llbracket E \rrbracket s).$$

In the valuation for assignment, the resulting state is passed on to other commands by sequencing or iteration of command meanings. In contrast, the new environment created by a definition is only used 'locally' within the body of the block.

As an example, let  $u$  be any environment then

$$\begin{aligned} & \llbracket \text{let } n \text{ be } 2 \text{ in } (\text{let } n \text{ be } n + 1 \text{ in } n) + n \rrbracket u \\ &= \llbracket (\text{let } n \text{ be } n + 1 \text{ in } n) + n \rrbracket (u \mid n \mapsto 2) \\ &= \llbracket \text{let } n \text{ be } n + 1 \text{ in } n \rrbracket (u \mid n \mapsto 2) + \llbracket n \rrbracket (u \mid n \mapsto 2) \\ &= \llbracket n \rrbracket (u \mid n \mapsto \llbracket n + 1 \rrbracket (u \mid n \mapsto 2)) + \llbracket n \rrbracket (u \mid n \mapsto 2) \\ &= \llbracket n \rrbracket (u \mid n \mapsto 3) + \llbracket n \rrbracket (u \mid n \mapsto 2) \\ &= 3 + 2 \\ &= 5 \end{aligned}$$

The local environment created by the inner **let** is not used for the evaluation of the occurrence of  $n$  that is outside the scope of that definition.

The following are some examples of equivalences that can be validated using the valuation for local-definition blocks:

$$\text{let } \iota \text{ be } \iota \text{ in } X \equiv X \equiv \text{let } \iota \text{ be } X \text{ in } \iota$$

$$\begin{aligned} & \text{let } \iota \text{ be } X \text{ in } PQ \\ & \equiv (\text{let } \iota \text{ be } X \text{ in } P) (\text{let } \iota \text{ be } X \text{ in } Q) \end{aligned}$$

$$\begin{aligned} & \text{let } \iota \text{ be } P \text{ in } (\text{let } \iota \text{ be } Q \text{ in } R) \\ & \equiv \text{let } \iota \text{ be } (\text{let } \iota \text{ be } P \text{ in } Q) \text{ in } R \end{aligned}$$

### 3.2 Function definitions

The language fragment described in Section 3.1 allows a programmer to use pre-defined functions and also to re-name them using the local-definition block, but does not provide a way for a programmer to define 'new' functions. We therefore introduce the following notation:

*Function definition:*

$$\frac{\begin{array}{c} [\iota_0: \theta_0] \\ \vdots \\ P: \theta \end{array} \quad \begin{array}{c} [\iota: \theta_0 \rightarrow \theta] \\ \vdots \\ Q: \theta' \end{array}}{\text{let } \iota(\iota_0: \theta_0) = P \text{ in } Q: \theta'}$$

This construction defines  $\iota$  to be the local name (within  $Q$ ) of a function defined by ‘formal parameter’  $\iota_0$  and ‘body’  $P$ . For example, the following block defines a function *Add2* that returns the result of adding 2 to its argument:

**let** *Add2*( $n$ : **val**[**nat**]) =  $n + 2$   
**in** ...

The semantic equation for this construction is as follows:

$$\llbracket \text{let } \iota(\iota_0: \theta_0) = P \text{ in } Q \rrbracket u = \llbracket Q \rrbracket (u \mid \iota \mapsto f)$$

where  $f: \llbracket \theta_0 \rrbracket \rightarrow \llbracket \theta \rrbracket$  is the function defined by

$$f(a) = \llbracket P \rrbracket (u \mid \iota_0 \mapsto a)$$

for all  $a \in \llbracket \theta_0 \rrbracket$ . Notice the environment used for  $P$ : the formal parameter  $\iota_0$  is bound to the value  $a$  (of some actual parameter), but other identifiers in the body are interpreted in the environment  $u$  for the *definition* of the function, rather than the environment for its *call*. For example, the function defined in

**let** *Addm*( $n$ : **val**[**nat**]) =  $n + m$   
**in** ...

returns the result of adding  $m$  to its argument, where  $m$  is determined in the context where the function is *defined*, not where it happens to be *used*. Occurrences of the function name  $\iota$  in  $P$  are *not* recursive uses of the function being defined; recursion will be discussed in the next section.

It is also convenient to introduce notation for defining ‘anonymous’ functions:

*Abstraction*:

$$\frac{\begin{array}{c} [\iota: \theta] \\ \vdots \\ P: \theta' \end{array}}{\lambda \iota: \theta. P: \theta \rightarrow \theta'}$$

This kind of construct is termed a *lambda expression*; it appears in several programming languages, including LISP and ALGOL 68, and the similar notation  $x \mapsto \dots x \dots$  is used in mathematics. The phrase  $\lambda \iota: \theta. P$  denotes the function that maps arguments  $a$  of type  $\theta$  into the meaning of  $P$  when  $\iota$  denotes  $a$ . For example,  $\lambda n: \text{val}[\text{nat}]. n + m$  denotes the numerical function that returns the result of adding  $m$  to its argument, so that the value of

$$(\lambda n: \text{val}[\text{nat}]. n + m)(3)$$

is that of  $3 + m$ . As usual, we assume that the syntactic scope of the lambda expression extends as far to the right as possible. The name of the

syntax rule, Abstraction, derives from the fact that the notation defines a function by abstracting from a particular syntactic representation of it.

The semantic equation for the lambda expression is

$$\begin{aligned} \llbracket \lambda \iota: \theta. P \rrbracket_{\pi(\theta \rightarrow \theta')}(u) &= f \in \llbracket \theta \rrbracket \rightarrow \llbracket \theta' \rrbracket \\ \text{such that } f(a) &= \llbracket P \rrbracket_{(\pi|_{\iota \mapsto \theta})\theta'}(u \mid \iota \mapsto a) \text{ for all } a \in \llbracket \theta \rrbracket. \end{aligned}$$

As with the function definition block, the values of free identifiers of the body  $P$  (other than the formal parameter  $\iota$ ) are determined in the environment of the  $\lambda$ -expression, rather than in the environments in which the function happens to be used.

### 3.3 Defined notation

It is straightforward to use the semantic equations of Sections 3.1 and 3.2 to prove that, in any appropriate phrase-type assignment, the function-definition block

$$\text{let } \iota(\iota_0: \theta_0) = P \text{ in } Q$$

and the simple definition block

$$\text{let } \iota \text{ be } \lambda \iota_0: \theta_0. P \text{ in } Q$$

are semantically equivalent for any  $P$  and  $Q$  (of appropriate type); furthermore, for any  $P$  of type  $\theta$  (in some type assignment), the simple definition block

$$\text{let } \iota \text{ be } P \text{ in } Q$$

and the function application

$$(\lambda \iota: \theta. Q)P$$

are semantically equivalent in that type assignment. We have essentially ‘reduced’ our two forms of local-definition block to particular combinations of functional abstraction and application. We can subsequently treat the two local-definition forms as *defined notation*, rather than as basic features of our language, much as we treated **if**  $B$  **then**  $C$  as an abbreviation for **if**  $B$  **then**  $C$  **else skip**.

As another example of this technique, we can reduce functions with multiple parameters to single-parameter functions. For any sets  $A, B$ , and  $C$ , there is a one-to-one correspondence between the set of two-argument functions  $A \times B \rightarrow C$  and the set of function-producing functions  $A \rightarrow (B \rightarrow C)$ : any  $f: A \times B \rightarrow C$  corresponds to  $g: A \rightarrow (B \rightarrow C)$  such that, for all  $a \in A$ ,  $g(a) = h_a \in B \rightarrow C$  such that  $h_a(b) = f(a, b)$  for all



$b \in B$ ; conversely,  $g: A \rightarrow (B \rightarrow C)$  corresponds to  $f: A \times B \rightarrow C$  such that for all  $a \in A$  and  $b \in B$ ,  $f(a, b) = g(a)(b)$ .

This correspondence allows us to regard a phrase type of the form

$$\theta_1 \times \theta_2 \times \cdots \times \theta_n \rightarrow \theta$$

as an alternative notation for

$$\theta_1 \rightarrow (\theta_2 \rightarrow (\cdots \rightarrow (\theta_n \rightarrow \theta) \cdots)),$$

which is essentially similar to the way 'multi-dimensional arrays' are treated in the programming language PASCAL: they are really arrays of arrays. We then can make the following syntactic extensions:

*Multiple application:*

$$\frac{P: \theta_1 \times \cdots \times \theta_n \rightarrow \theta \quad Q_i: \theta_i \text{ for } i = 1, 2, \dots, n}{P(Q_1, \dots, Q_n): \theta}$$

*Multiple abstraction:*

$$\frac{\begin{array}{c} \left[ \begin{array}{c} \iota_1: \theta_1 \\ \vdots \\ \iota_n: \theta_n \end{array} \right] \\ \vdots \\ P: \theta \end{array}}{\lambda(\iota_1: \theta_1, \dots, \iota_n: \theta_n). P: \theta_1 \times \cdots \times \theta_n \rightarrow \theta}$$

*Multiple-parameter function definition:*

$$\frac{\begin{array}{c} \left[ \begin{array}{c} \iota_1: \theta_1 \\ \vdots \\ \iota_n: \theta_n \end{array} \right] \\ \vdots \\ P: \theta \end{array} \quad \begin{array}{c} [\iota: \theta_1 \times \cdots \times \theta_n \rightarrow \theta] \\ \vdots \\ Q: \theta' \end{array}}{\text{let } \iota(\iota_1: \theta_1, \dots, \iota_n: \theta_n) = P \text{ in } Q: \theta'}$$

The interpretations can be defined by the following equivalences:

$$P(Q_1, \dots, Q_n) \equiv P(Q_1) \cdots (Q_n)$$

$$\lambda(\iota_1: \theta_1, \dots, \iota_n: \theta_n). P \equiv \lambda \iota_1: \theta_1. \cdots \lambda \iota_n: \theta_n. P$$

$$\begin{aligned} & \text{let } \iota(\iota_1: \theta_1, \dots, \iota_n: \theta_n) = P \text{ in } Q \\ & \equiv (\lambda \iota: \theta_1 \times \dots \times \theta_n \rightarrow \theta. Q)(\lambda(\iota_1: \theta_1, \dots, \iota_n: \theta_n). P), \end{aligned}$$

where we have omitted the obvious type subscripts on  $\equiv$ . Variants of the **let** forms with *multiple* definitions could be introduced in the same way.

Another example of defined notation is based on the notation for quantification in predicate logic:

*Quantification:*

$$\frac{\begin{array}{c} [\iota: \theta'] \\ \vdots \\ Q: (\theta' \rightarrow \theta'') \rightarrow \theta \quad P: \theta'' \end{array}}{\# Q \iota. P: \theta}$$

Phrase  $Q$  is the *quantifier*,  $\iota$  is the bound identifier, and  $P$  is the body. The interpretation is defined by the following equivalence:

$$\# Q \iota. P \equiv_{\theta} Q(\lambda \iota: \theta'. P);$$

that is, for any appropriate environment  $u$ ,

$$\llbracket \# Q \iota. P \rrbracket u = \llbracket Q \rrbracket u f,$$

where  $f(a) = \llbracket P \rrbracket (u \mid \iota \mapsto a)$  for all  $a \in \llbracket \theta' \rrbracket$ . For example, a programmer can obtain the analogue of

$$\sum_{i=a}^b \dots i \dots$$

by using a suitable higher-order function

$$\text{sum}: \text{val}[\text{nat}] \times \text{val}[\text{nat}] \rightarrow (\text{val}[\text{nat}] \rightarrow \text{val}[\text{nat}]) \rightarrow \text{val}[\text{nat}]$$

as follows:

$$\# \text{sum}(a, b) i. \dots i \dots.$$

This is by definition equivalent to

$$\text{sum}(a, b, \lambda i: \text{val}[\text{nat}]. \dots i \dots),$$

but the advantages of the quantification notation are that the syntactic structure is clearer and the type of the bound identifier need not be stated explicitly (because it can be inferred from the type of the quantifier).

We will use the quantification notation to introduce in a uniform way a variety of language features involving identifier binding.

### 3.4 Elementary properties

The applicative language described in the preceding three sub-sections is termed an *explicitly-typed* language because it has the following two properties.

1. For any phrase  $X$  and phrase-type assignment  $\pi$ ,  $\pi \vdash X: \theta$  is derivable for at most *one* phrase type  $\theta$ .
2. If it is defined at all, the type of any composite phrase  $X$  in a phrase-type assignment  $\pi$  is determined by the types of the immediate constituents of  $X$  in some determined phrase-type assignments (possibly, but not necessarily  $\pi$ ).

For example, the type of  $\lambda\iota: \theta. X$  in  $\pi$  is defined just if  $X$  has a type in  $(\pi \mid \iota \mapsto \theta)$ , and is then  $\theta \rightarrow \theta'$ , where  $\theta'$  is the *unique* type of  $X$  in that phrase-type assignment. If the syntax did not require that the type of  $\iota$  be specified explicitly, a lambda expression could have many types; e.g.,  $\lambda x. x$  has type  $\theta \rightarrow \theta$  for *every*  $\theta$ .

As another example, the type of **let**  $\iota$  **be**  $P$  **in**  $Q$  in phrase-type assignment  $\pi$  is defined just if  $P$  has a type in  $\pi$  and  $Q$  has a type in  $(\pi \mid \iota \mapsto \theta)$ , where  $\theta$  is *the* type of  $P$  in  $\pi$ , and is then *the* type of  $Q$  in  $(\pi \mid \iota \mapsto \theta)$ . The explicit-typing property makes it possible to *infer* the type of  $\iota$  from the type of  $P$ , despite the fact that there is no *explicit* occurrence of the type of  $\iota$  in the construction.

As a final example, consider the function-definition block

$$\text{let } \iota(\iota_0: \theta_0) = P \text{ in } Q$$

The explicit occurrence of the argument type  $\theta_0$  is here essential to determine the phrase-type assignment used for  $P$ . On the other hand, the result type  $\theta$  can be inferred from  $P$ .

We have seen that, given  $\pi$ , there is at most one  $\theta$  such that  $\pi \vdash X: \theta$  is derivable; but given some phrase type  $\theta$ , it is not the case that there is at most one phrase-type assignment  $\pi$  such that  $\pi \vdash X: \theta$ . Let  $\pi' \vdash \pi$  just if  $\pi'$  is an extension of  $\pi$ ; i.e.,  $\pi' \vdash \pi$  if and only if  $\text{dom } \pi \subseteq \text{dom } \pi'$  and, for all  $\iota \in \text{dom } \pi$ ,  $\pi(\iota) = \pi'(\iota)$ .

**Proposition 3.4.1.** *If  $\pi' \vdash \pi$  and  $\pi \vdash X: \theta$  is derivable then so is  $\pi' \vdash X: \theta$ .*

**Proof.** By induction on the length of derivations of sequents in the formal system for syntax. If  $\pi \vdash \iota: \pi(\iota)$  is derivable, then so is  $\pi' \vdash \iota: \pi(\iota)$  because  $\pi'(\iota) = \pi(\iota)$ .

If  $\pi \vdash \lambda\iota: \theta. P: \theta \rightarrow \theta'$  is derivable, then this follows directly from the derivability of  $(\pi \mid \iota \mapsto \theta) \vdash P: \theta'$ ; but  $(\pi' \mid \iota \mapsto \theta)$  is an extension of  $(\pi \mid \iota \mapsto \theta)$  and so, by induction,  $(\pi' \mid \iota \mapsto \theta) \vdash P: \theta'$  is derivable and then it follows that  $\pi' \vdash \lambda\iota: \theta. P: \theta \rightarrow \theta'$  is derivable.

If  $\pi \vdash \text{not } B: \text{val}[\text{bool}]$  is derivable, then this follows directly from the derivability of  $\pi \vdash B: \text{val}[\text{bool}]$ ; by induction,  $\pi' \vdash B: \text{val}[\text{bool}]$  is derivable, so that  $\pi' \vdash \text{not } B: \text{val}[\text{bool}]$  is also, and similarly for other composite phrases. ■

This syntactic ‘ambiguity’ raises the possibility that a phrase might have significantly different semantic interpretations, depending on which ‘unnecessary’ assumptions happened to be made in the derivation of its syntactic well-formedness. The following result shows that this is not the case.

**Lemma 3.4.2 (Coherence).** *If  $\pi \vdash X: \theta$  and  $\pi' \vdash \pi$ , then, for every  $u' \in \llbracket \pi' \rrbracket$ ,*

$$\llbracket X \rrbracket_{\pi' \theta}(u') = \llbracket X \rrbracket_{\pi \theta}(u),$$

where  $u \in \llbracket \pi \rrbracket$  is the restriction of  $u'$  to  $\text{dom } \pi$ .

If we use the notation  $\llbracket \pi' \vdash \pi \rrbracket$  to denote the function from  $\llbracket \pi' \rrbracket$  to  $\llbracket \pi \rrbracket$  that restricts any  $u' \in \llbracket \pi' \rrbracket$  to  $\text{dom } \pi$ , the coherence lemma can be expressed as the commutativity of the following diagram:

$$\begin{array}{ccc} \llbracket \pi' \rrbracket & & \\ \downarrow \llbracket \pi' \vdash \pi \rrbracket & \searrow \llbracket X \rrbracket_{\pi' \theta} & \\ \llbracket \pi \rrbracket & \xrightarrow{\llbracket X \rrbracket_{\pi \theta}} & \llbracket \theta \rrbracket \end{array}$$

**Proof.** By structural induction on the syntax of  $X$ . We discuss only the case that  $X = \lambda \iota: \theta_0. P$ . Suppose that

$$\pi \vdash \lambda \iota: \theta_0. P: \theta_0 \rightarrow \theta_1$$

is derivable and that  $\pi'$  is an extension of  $\pi$ ; then  $(\pi' \mid \iota \mapsto \theta_0)$  is an extension of  $(\pi \mid \iota \mapsto \theta_0)$ . Consider any  $u' \in \llbracket \pi' \rrbracket$  and let  $u \in \llbracket \pi \rrbracket$  be its restriction to  $\text{dom } \pi$ ; then, for any  $a \in \llbracket \theta_0 \rrbracket$ ,  $(u \mid \iota \mapsto a)$  is the restriction of  $(u' \mid \iota \mapsto a)$  to  $\text{dom}(\pi \mid \iota \mapsto \theta_0)$ , and so

$$\begin{aligned} \llbracket \lambda \iota: \theta_0. P \rrbracket u'a &= \llbracket P \rrbracket (u' \mid \iota \mapsto a) \\ &= \llbracket P \rrbracket (u \mid \iota \mapsto a) \quad (\text{by induction}) \\ &= \llbracket \lambda \iota: \theta_0. P \rrbracket ua \end{aligned}$$

The *minimal*  $\text{dom } \pi$  for which it is possible for a phrase  $X$  to be well-formed is the set  $\text{free}(X)$  of all identifiers that occur ‘unbound’ in  $X$ . This can be defined by induction on the structure of  $X$  as follows:

- (i)  $\text{free}(\iota) = \{\iota\}$  ;

- (ii)  $free(0) = \emptyset$ , and similarly for the remaining basic phrases;
- (iii)  $free(\lambda\iota: \theta. P) = free(P) - \{\iota\}$  ;
- (iv)  $free(P Q) = free(P) \cup free(Q)$ , and similarly for the remaining composite phrases that do not involve identifier binding.

The definitions of  $free(\cdot)$  for the other phrase forms that involve identifier binding can be derived using the defining equivalences for those constructs, as in the following:

$$\begin{aligned}
 free(\text{let } \iota \text{ be } P \text{ in } Q) \\
 &= free((\lambda\iota: \theta. Q)(P)) \\
 &= (free(Q) - \{\iota\}) \cup free(P)
 \end{aligned}$$

For example,

$$\begin{aligned}
 free(\text{let } m \text{ be } m + n \text{ in } m + n) \\
 &= (free(m + n) - \{m\}) \cup free(m + n) \\
 &= \{n\} \cup \{m, n\} \\
 &= \{m, n\}
 \end{aligned}$$

Notice that identifier  $m$  is both free and bound in this phrase.

Similarly

$$\begin{aligned}
 free(\text{let } \iota(\iota_0: \theta_0) = P \text{ in } Q) \\
 &= free((\lambda\iota: \theta_0 \rightarrow \theta. Q)(\lambda\iota_0: \theta_0. P)) \\
 &= (free(Q) - \{\iota\}) \cup (free(P) - \{\iota_0\}).
 \end{aligned}$$

The following proposition shows that  $free(X)$  as defined above is neither too large, nor too small.

**Proposition 3.4.3.**

- (a) If  $\pi \vdash X: \theta$  then  $free(X) \subseteq \text{dom } \pi$ .
- (b) If  $\pi \vdash X: \theta$  and  $\pi'$  is the restriction of  $\pi$  to  $free(X)$  then  $\pi' \vdash X: \theta$ .

**Proof.** Each part can be proved by induction on the syntax of  $X$ . We discuss only the case that  $X = P Q$ . Assume that  $\pi \vdash P Q: \theta'$  because  $\pi \vdash P: \theta \rightarrow \theta'$  and  $\pi \vdash Q: \theta$ .

- (a) By induction,  $free(P) \subseteq \text{dom } \pi$  and  $free(Q) \subseteq \text{dom } \pi$ , and so

$$free(P Q) = free(P) \cup free(Q) \subseteq \text{dom } \pi,$$

as desired.

- (b) Let  $\pi_1$  be the restriction of  $\pi$  to  $free(P)$  and  $\pi_2$  be the restriction of  $\pi$  to  $free(Q)$ ; by induction,  $\pi_1 \vdash P: \theta \rightarrow \theta'$  and  $\pi_2 \vdash Q: \theta$ . Let  $\pi'$  be



the restriction of  $\pi$  to  $\text{free}(PQ)$ ;  $\pi'$  is an extension of both  $\pi_1$  and  $\pi_2$ , so that, by Proposition 3.4.1,  $\pi' \vdash P: \theta \rightarrow \theta'$  and  $\pi' \vdash Q: \theta$ , and so  $\pi' \vdash PQ: \theta'$ , as desired. ■

Combining the coherence Lemma with this proposition gives us the following useful fact.

**Proposition 3.4.4.** *For any phrase  $X$ ,  $\llbracket X \rrbracket_{u_1} = \llbracket X \rrbracket_{u_2}$  whenever environments  $u_1$  and  $u_2$  agree on  $\text{free}(X)$ .*

**Proof.** By Proposition 3.4.3,  $X$  is well-formed in the phrase-type assignment obtained by restricting to  $\text{free}(X)$  and, by the coherence lemma, each side is equal to  $\llbracket X \rrbracket_u$ , where  $u$  is the restriction of  $u_1$  and  $u_2$  to  $\text{free}(X)$ . ■

## 3.5 Programming logic

In this sub-section, we describe a formal system for reasoning about semantic equivalences in our applicative language.

### 3.5.1 Syntax and semantics

We take the following formulas as the atomic specifications:

*Equivalence:*

$$\frac{X_0: \theta \quad X_1: \theta}{X_0 \equiv_{\theta} X_1: \mathbf{spec}}$$

We have been using equivalences in our meta-language; now, they are part of our formal object language as well. We adopt the propositional connectives  $\&$  (conjunction) and  $\Rightarrow$  (implication), and also a propositional constant **absurd**, and

*Universal quantification:*

$$\frac{\begin{array}{c} [\iota: \theta] \\ \vdots \\ Z: \mathbf{spec} \end{array}}{\forall \iota: \theta. Z: \mathbf{spec}}$$

Intuitively,  $\forall \iota: \theta. Z$  asserts that  $Z$  is *true* for  $\iota$  denoting any element of  $\llbracket \theta \rrbracket$ . Note that the syntax rule is in natural-deduction format, and all the occurrences of  $\iota$  in formula  $\forall \iota: \theta. Z$  are bound;  $\text{free}(\forall \iota: \theta. Z)$  can be defined by analogy with  $\lambda \iota: \theta. X$ , which has the same identifier-binding structure.

Semantically, we have  $\llbracket \mathbf{spec} \rrbracket = \{\text{true}, \text{false}\}$ , as before, but the interpretation of specification formulas must now use environments to allow

determination of the meanings of the free identifiers; hence we define valuations

$$\llbracket \cdot \rrbracket_{\pi \text{ spec}}: [\text{spec}]_{\pi} \rightarrow (\llbracket \pi \rrbracket \rightarrow \llbracket \text{spec} \rrbracket)$$

as follows:

$$\llbracket \forall \iota: \theta. Z \rrbracket u = \text{for all } m \in \llbracket \theta \rrbracket, \llbracket Z \rrbracket(u \mid \iota \mapsto m)$$

$$\llbracket \text{absurd} \rrbracket u = \text{false}$$

$$\llbracket Z_0 \& Z_1 \rrbracket u = \begin{cases} \text{true,} & \text{if } \llbracket Z_0 \rrbracket u = \text{true and } \llbracket Z_1 \rrbracket u = \text{true} \\ \text{false,} & \text{otherwise} \end{cases}$$

and similarly for the implication connective, and

$$\llbracket X_0 \equiv X_1 \rrbracket u = (\llbracket X_0 \rrbracket u = \llbracket X_1 \rrbracket u).$$

This completes the description of the syntax and semantics of the programming logic for our applicative language.

### 3.5.2 Substitution

In Section 2.6.2, we introduced the meta-linguistic notation  $[P](\iota \mapsto E)$  to denote the result of substituting  $E$  for all occurrences of a variable-identifier  $\iota$  in  $P$  and this concept played an important role in the programming logic for the simple imperative language. Substitution is also important in reasoning about programs in our applicative language, but we must first generalize the definition of substitution in several ways.

One generalization is that we must allow substitutions for *any* type of identifier, not just the variable-identifiers of Section 2. However, we must take into account identifier-binding constructions like the lambda expression; we do not want to substitute for *locally bound* occurrences of identifiers. For example,

$$[\lambda n: \text{val}[\text{nat}]. n + 1](n \mapsto a + 1)$$

should be equal to  $\lambda n: \text{val}[\text{nat}]. n + 1$ ; the occurrences of  $n$  in the lambda expression are to be regarded as occurrences of a different  $n$  than is being substituted for.

A further complication arises if an identifier is *both* free in a substituted phrase *and* locally bound in the context of the substituted identifier; we do not want such a local binding to ‘capture’ the free identifier of the substituted phrase. For example,

$$[\lambda n: \text{val}[\text{nat}]. n + m](m \mapsto n + 2)$$

should *not* be  $\lambda n: \text{val}[\text{nat}]. n + (n + 2)$ ; the occurrence of  $n$  in the substituted phrase  $n + 2$  is to be regarded as an occurrence of a different  $n$  than the formal parameter of the lambda expression. A solution to this

problem is to take advantage of the fact that the choice of bound identifier is semantically irrelevant by changing all bound occurrences of the local identifier to another identifier that does not appear free in the substituted phrase. For example, if we replace the bound identifier  $n$  by  $k$ , we obtain a semantically equivalent phrase on which the substitution can be carried out without difficulty:

$$\begin{aligned} & [\lambda n: \text{val}[\text{nat}]. n + m](m \mapsto n + 2) \\ & \equiv [\lambda k: \text{val}[\text{nat}]. k + m](m \mapsto n + 2) \\ & = \lambda k: \text{val}[\text{nat}]. k + (n + 2). \end{aligned}$$

Finally, it is technically convenient to treat simultaneous *multiple* substitution (of several identifiers) as the basic concept (with single-identifier substitution as a special case).

Substitutions will be specified by *phrase assignments*; i.e., functions from finite sets of identifiers to phrases, thought of as specifying simultaneous multiple substitutions of the phrases for the identifiers. If  $\sigma$  is a phrase assignment, let

$$\text{free}(\sigma) = \bigcup_{\iota \in \text{dom } \sigma} \text{free}(\sigma(\iota)).$$

For any phrase  $X$  and phrase assignment  $\sigma$  with  $\text{dom } \sigma \supseteq \text{free}(X)$ , we define  $[X]\sigma$ , the result of carrying out the substitutions specified by  $\sigma$  on  $X$ , by induction on the syntax of  $X$  as follows:

- (i)  $[\iota]\sigma = \sigma(\iota)$ ;
- (ii)  $[0]\sigma = 0$ , and similarly for the remaining basic phrases;
- (iii)  $[\lambda \iota_0: \theta. P]\sigma = \lambda \iota_1: \theta. [P](\sigma \mid \iota_0 \mapsto \iota_1)$ , where  $\iota_1$  can be any identifier not in  $\text{free}(\sigma)$  (thereby precluding ‘capture’ of free occurrences of  $\iota_0$  in the substituted phrases), and similarly for  $[\forall \iota: \theta. Z]\sigma$ ;
- (iv)  $[PQ]\sigma = ([P]\sigma) ([Q]\sigma)$ , and similarly for the remaining composite phrases that do not involve identifier binding.

Strictly speaking,  $[X]\sigma$  is not uniquely determined because of the choice allowed for identifier  $\iota_1$  in case (iii). If uniqueness is desired, it is necessary to select that identifier in some determinate way; for example, it is possible to order the set of identifiers (say, alphabetically) and then select the minimal identifier (according to this ordering) not free in  $\sigma$ .

The definition of substitution for the other phrase forms that involve identifier binding can be derived using the defining equivalences for those constructs, as in the following:

$$\begin{aligned} & [\text{let } \iota_0 \text{ be } P \text{ in } Q]\sigma \\ & = [(\lambda \iota_0: \theta. Q)(P)]\sigma \\ & = (\lambda \iota_1: \theta. [Q](\sigma \mid \iota_0 \mapsto \iota_1)) ([P]\sigma) \end{aligned}$$

$$= \text{let } \iota_1 \text{ be } [P]\sigma \text{ in } [Q](\sigma \mid \iota_0 \mapsto \iota_1),$$

where  $\iota_1$  can be any identifier not in  $\text{free}(\sigma)$ . For example, for any phrase assignment  $\sigma$ ,

$$\begin{aligned} & [\text{let } m \text{ be } m + n \text{ in } m + n](\sigma \mid m \mapsto m \mid n \mapsto m + k) \\ &= \text{let } m' \text{ be } m + (m + k) \text{ in } m' + (m + k), \end{aligned}$$

where  $m'$  is any identifier different from  $m$  and  $k$ .

**Exercise 3.5.1.** Define how to do substitutions into the function-definition form of block.

In Section 2, we used substitutions only when substituting into a phrase would preserve its type. We still want substitutions to preserve types, but here the situation is more complicated because the type of a phrase can only be determined in a given type assignment to its free identifiers. For any phrase-type assignments  $\pi_0$  and  $\pi_1$ , let  $[\pi_0]_{\pi_1}$  denote the following set of *type-preserving* phrase assignments:

$$\prod_{\iota \in \text{dom } \pi_0} [\pi_0(\iota)]_{\pi_1};$$

that is,  $\sigma \in [\pi_0]_{\pi_1}$  is required to be a function with the same domain as  $\pi_0$  that maps each identifier  $\iota \in \text{dom } \pi_0$  into a phrase  $\sigma(\iota)$  such that, if  $\pi_0(\iota) = \theta$ , then  $\pi_1 \vdash \sigma(\iota) : \theta$ .

For example, suppose that

$$\begin{aligned} \pi_0 &= \{n: \text{val}[\text{nat}], b: \text{val}[\text{bool}]\}, \text{ and} \\ \pi_1 &= \{m: \text{val}[\text{nat}]\}, \end{aligned}$$

and let  $\sigma$  be the phrase assignment such that

- $\text{dom } \sigma = \{n, b\}$ ,
- $\sigma(n) = (m + 1)$ , and
- $\sigma(b) = (m > 0)$ ;

then  $\sigma \in [\pi_0]_{\pi_1}$  because  $(m+1) \in [\text{val}[\text{nat}]]_{\pi_1}$  and  $(m > 0) \in [\text{val}[\text{bool}]]_{\pi_1}$ .

The following proposition shows that our definition of substitution is syntactically sound in the sense that if  $\pi_0$  is the phrase-type assignment for a phrase *before* a substitution,  $\pi_1$  is the phrase-type assignment for the phrase *after* the substitution, and  $\sigma$  preserves types (with respect to  $\pi_0$  and  $\pi_1$ ), then any substitution specified by  $\sigma$  preserves the type of the phrase as a whole (with respect to  $\pi_0$  and  $\pi_1$ ).

**Proposition 3.5.2.** *If  $X \in [\theta]_{\pi_0}$  and  $\sigma \in [\pi_0]_{\pi_1}$ , then  $[X]\sigma \in [\theta]_{\pi_1}$ .*

The phrase type  $\theta$  here (and in the following lemma) may be **spec** as well as any of the ordinary phrase types.

**Proof.** By structural induction on the syntax of  $X$ . We discuss only the case that  $X = \lambda\iota_0: \theta. P$ . If  $\lambda\iota_0: \theta. P \in [\theta \rightarrow \theta']_{\pi_0}$  then  $P \in [\theta']_{(\pi_0|\iota_0 \mapsto \theta)}$ . We first verify that

$$(\sigma \mid \iota_0 \mapsto \iota_1) \in [(\pi_0 \mid \iota_0 \mapsto \theta)]_{(\pi_1|\iota_1 \mapsto \theta)}$$

for any  $\iota_1 \notin \text{free}(\sigma)$ ; we need to show that, for all  $\iota \in (\text{dom } \pi_0 \cup \{\iota_0\})$ ,

$$(\pi_1 \mid \iota_1 \mapsto \theta) \vdash (\sigma \mid \iota_0 \mapsto \iota_1)(\iota) : (\pi_0 \mid \iota_0 \mapsto \theta)(\iota).$$

If  $\iota = \iota_0$ ,  $(\pi_1 \mid \iota_1 \mapsto \theta) \vdash \iota_1 : \theta$ ; if  $\iota \neq \iota_0$ ,  $(\pi_1 \mid \iota_1 \mapsto \theta) \vdash \sigma(\iota) : \pi_0(\iota)$  because  $\sigma \in [\pi_0]_{\pi_1}$  and  $\iota_1 \notin \text{free}(\sigma)$ . By induction,

$$[P](\sigma \mid \iota_0 \mapsto \iota_1) \in [\theta']_{(\pi_1|\iota_1 \mapsto \theta)},$$

and so

$$\lambda\iota_1: \theta. [P](\sigma \mid \iota_0 \mapsto \iota_1) \in [\theta \rightarrow \theta']_{\pi_1},$$

as desired. ■

The next result shows that our definition of substitution is also *semantically* sound, in the sense that the meaning of a phrase obtained by doing the substitution specified by a phrase assignment  $\sigma$  is that of the *original* phrase in a certain environment definable from  $\sigma$ . For reasoning about program meanings in concrete applications, syntactic substitution is an essential technique; but for reasoning about properties of the language as a whole, the corresponding semantic definition is more convenient. The following lemma assures us that the definition of substitution is sound relative to the semantic interpretation.

**Lemma 3.5.3 (Substitution).** *If  $X \in [\theta]_{\pi_0}$ ,  $\sigma \in [\pi_0]_{\pi_1}$ , and  $u_1 \in \llbracket \pi_1 \rrbracket$ , then*

$$\llbracket [X]\sigma \rrbracket_{\pi_1\theta}(u_1) = \llbracket X \rrbracket_{\pi_0\theta}(u_0),$$

where  $u_0 \in \llbracket \pi_0 \rrbracket$  is defined as follows: for all  $\iota \in \text{dom } \pi_0$ ,

$$u_0(\iota) = \llbracket \sigma(\iota) \rrbracket_{\pi_1 \pi_0(\iota)}(u_1).$$

As a special case of the Substitution lemma, let  $\pi_0 = (\pi_1 \mid \iota \mapsto \theta')$ ,  $P \in [\theta']_{\pi_1}$ , and  $\sigma \in [\pi_1]_{\pi_1}$  map every identifier in  $\text{dom } \pi_1$  to itself; then

$$\llbracket [X](\sigma \mid \iota \mapsto P) \rrbracket_{\pi_1\theta}(u_1) = \llbracket X \rrbracket_{\pi_0\theta}(u_1 \mid \iota \mapsto \llbracket P \rrbracket_{\pi_1\theta'}(u_1)),$$



which is comparable to (but much more general than) Proposition 2.6.2 in Section 2.6.3.

**Proof.** The Substitution lemma can be proved by structural induction on  $X$ . We discuss only the lambda-expression case in detail. Omitting subscripts on valuation brackets, we must show that

$$\llbracket [\lambda \iota_0: \theta. P] \sigma \rrbracket u_1 a = \llbracket [\lambda \iota_0: \theta. P] u_0 a \rrbracket$$

for all  $a \in \llbracket \theta \rrbracket$ , where  $u_0(\iota) = \llbracket \sigma(\iota) \rrbracket u_1$  for all  $\iota \in \text{dom } \pi_0$ . The left-hand side equals

$$\begin{aligned} & \llbracket [\lambda \iota_1: \theta. [P](\sigma \mid \iota_0 \mapsto \iota_1)] u_1 a \rrbracket \quad (\text{for } \iota_1 \notin \text{free}(\sigma)) \\ &= \llbracket [P](\sigma \mid \iota_0 \mapsto \iota_1) \rrbracket (u_1 \mid \iota_1 \mapsto a) \\ &= \llbracket P \rrbracket u'_0, \end{aligned}$$

where  $u'_0(\iota) = \llbracket (\sigma \mid \iota_0 \mapsto \iota_1)(\iota) \rrbracket (u_1 \mid \iota_1 \mapsto a)$  (by induction). The right-hand side equals  $\llbracket P \rrbracket (u_0 \mid \iota_0 \mapsto a)$ , and so we need only show that

$$u'_0(\iota) = (u_0 \mid \iota_0 \mapsto a)(\iota)$$

for all  $\iota \in (\text{dom } \pi_0 \cup \{\iota_0\})$ . If  $\iota = \iota_0$ , both sides are equal to  $a$ ; if  $\iota \neq \iota_0$ , both sides are equal to  $\llbracket \sigma(\iota) \rrbracket u_1$ , using Proposition 3.4.4 and the fact that  $\iota_1 \notin \text{free}(\sigma(\iota))$ . ■

**Exercise 3.5.4.** Define, for any phrase-type assignments  $\pi_0$  and  $\pi_1$ , a substitution valuation function

$$\llbracket \cdot \rrbracket_{\pi_1 \pi_0}: [\pi_0]_{\pi_1} \rightarrow ([\pi_1] \rightarrow [\pi_0])$$

such that, for any  $\sigma \in [\pi_0]_{\pi_1}$  and  $X \in [\theta]_{\pi_0}$ ,

$$\llbracket [X] \sigma \rrbracket_{\pi_1 \theta} = \llbracket \sigma \rrbracket_{\pi_1 \pi_0}; \llbracket X \rrbracket_{\pi_0 \theta}$$

As a simple first application of the Substitution lemma, we can verify that, for any  $P \in [\theta]_{\pi}$  and  $Q \in [\theta']_{(\pi \mid \iota \mapsto \theta)}$ ,

$$\text{let } \iota \text{ be } P \text{ in } Q$$

is semantically equivalent in phrase-type assignment  $\pi$  to

$$[Q](\sigma \mid \iota \mapsto P),$$

where  $\sigma$  maps every identifier in  $\text{dom } \pi$  to itself. Consider any  $u \in \llbracket \pi \rrbracket$ ; then

$$\llbracket \text{let } \iota \text{ be } P \text{ in } Q \rrbracket u$$

and

$$\llbracket [Q](\sigma \mid \iota \mapsto P) \rrbracket u$$

are both equal to

$$\llbracket Q \rrbracket (u \mid \iota \mapsto \llbracket P \rrbracket u),$$

by the semantic equation for **let** and the Substitution lemma, respectively.

We will regard

$$[Q](\iota \mapsto P)$$

as an abbreviation of  $[Q](\sigma \mid \iota \mapsto P)$ , where  $\sigma$  is a phrase assignment that maps every identifier free in  $Q$  to itself, and similarly for

$$[Q](\iota_1 \mapsto P_1 \mid \cdots \mid \iota_n \mapsto P_n).$$

### 3.5.3 Axioms

We are finally ready to present the axioms for our programming logic. These will be presented as sequents of the form  $\pi \vdash Z$ , with  $Z \in [\text{spec}]_\pi$ . We start with the general properties of equivalences.

*Reflexivity:*

$$\pi \vdash X \equiv_\theta X,$$

when  $X \in [\theta]_\pi$ .

*Transitivity:*

$$\pi \vdash X_0 \equiv_\theta X_1 \ \& \ X_1 \equiv_\theta X_2 \Rightarrow X_0 \equiv_\theta X_2,$$

when  $X_0, X_1, X_2 \in [\theta]_\pi$ .

*Symmetry:*

$$\pi \vdash X_0 \equiv_\theta X_1 \Rightarrow X_1 \equiv_\theta X_0,$$

when  $X_0, X_1 \in [\theta]_\pi$ .

*Substitutivity:*

$$\begin{aligned} \pi \vdash X_1 \equiv_{\theta_1} Q_1 \ \& \ \cdots \ \& \ X_n \equiv_{\theta_n} Q_n \ \& \ [Z](\iota_1 \mapsto X_1 \mid \cdots \mid \iota_n \mapsto X_n) \\ \Rightarrow [Z](\iota_1 \mapsto Q_1 \mid \cdots \mid \iota_n \mapsto Q_n), \end{aligned}$$

when  $X_i, Q_i \in [\theta_i]_\pi$ ,  $Z \in [\text{spec}]_{(\pi \mid \cdots \mid \iota_i \mapsto \theta_i \mid \cdots)}$ .

Mathematical facts about the data types lead to domain-specific axioms; for example, we can have the following axiom:

$$\pi \vdash \forall n: \text{val}[\text{nat}]. \text{succ } n \neq_{\text{val}[\text{nat}]} 0,$$

where  $X \neq_{\theta} X'$  is an abbreviation for  $X \equiv_{\theta} X' \Rightarrow \mathbf{absurd}$ .

Reasoning about phrases of *functional* type can be based on the following axioms:

*Extensionality*:

$$\pi \vdash (\forall \iota: \theta. F_0(\iota) \equiv_{\theta'} F_1(\iota)) \iff F_0 \equiv_{\theta \rightarrow \theta'} F_1,$$

when  $F_0, F_1 \in [\theta \rightarrow \theta']_{\pi}$  and  $\iota \notin \text{dom } \pi$ .

*Alpha*:

$$\pi \vdash \lambda \iota_0: \theta. P \equiv_{\theta \rightarrow \theta'} \lambda \iota_1: \theta. [P](\iota_0 \mapsto \iota_1),$$

when  $P \in [\theta']_{(\pi | \iota_0 \mapsto \theta)}$  and  $\iota_1 \notin \text{dom } \pi$ .

*Beta*:

$$(\pi | \iota \mapsto \theta) \vdash (\lambda \iota: \theta. Q)(\iota) \equiv_{\theta'} Q,$$

when  $Q \in [\theta']_{(\pi | \iota \mapsto \theta)}$ .

*Eta*:

$$\pi \vdash \lambda \iota: \theta. F(\iota) \equiv_{\theta \rightarrow \theta'} F,$$

when  $F \in [\theta \rightarrow \theta']_{\pi}$  and  $\iota \notin \text{dom } \pi$ .

The Extensionality law states that the meanings of phrases of functional types are fully determined by their applicative behaviour. The Alpha law justifies ‘systematic changes of identifier’, as in the definition of substitution; and the Beta and Eta laws assert that abstracting with respect to an identifier  $\iota$  and applying to  $\iota$  are inverse operations. The names of these axioms are those of the corresponding axioms in the (simply-typed) lambda calculus, which is the purely equational form of this system.

The validity of all of these axioms can be verified using the semantic valuations; we discuss the Alpha law. Consider any  $u \in \llbracket \pi \rrbracket$  and  $a \in \llbracket \theta \rrbracket$ , and a phrase assignment  $\sigma$  that maps every element of  $\text{dom } \pi \cup \{\iota_1\}$  to itself; then

$$\begin{aligned} & \llbracket \lambda \iota_1: \theta. [P](\sigma | \iota_0 \mapsto \iota_1) \rrbracket u a \\ &= \llbracket [P](\sigma | \iota_0 \mapsto \iota_1) \rrbracket (u | \iota_1 \mapsto a) \\ &= \llbracket [P](u | \iota_1 \mapsto a | \iota_0 \mapsto a) \rrbracket \quad (\text{by the Substitution lemma}) \\ &= \llbracket [P](u | \iota_0 \mapsto a) \rrbracket \quad (\text{by the Coherence lemma, because } \iota_1 \notin \text{dom } \pi) \\ &= \llbracket \lambda \iota_0: \theta. P \rrbracket u a \end{aligned}$$

We assume standard (intuitionistic) rules for the logical connectives and quantifiers. Further equivalences can then be *derived* for defined notation; for example, the following derivation

$$\frac{\frac{\frac{[\iota: \theta]}{(\lambda\iota: \theta. Q)(\iota) \equiv_{\theta'} Q} \text{ (Beta)}}{\forall\iota: \theta. (\lambda\iota: \theta. Q)(\iota) \equiv_{\theta'} Q} \text{ (\forall-I)}}{[(\lambda\iota: \theta. Q)(\iota) \equiv_{\theta'} Q](\iota \mapsto P)} \text{ (\forall-E)}$$

uses Beta and the usual rules of universal-quantifier introduction ( $\forall$ -I) and elimination ( $\forall$ -E) to give us the equivalence of  $[Q](\iota \mapsto P)$  and

$$\text{let } \iota \text{ be } P \text{ in } Q$$

that was verified semantically in Section 3.5.2.

### 3.6 Bibliographic notes

The essential features of the language discussed in this section were first described in [Church, 1940]; modern presentations may be found in [Mitchell, 1990] and [Barendregt, 1992]. The treatment of multi-argument functions as function-producing functions is due to [Schönfinkel, 1924], but the technique is often called ‘Currying’, after [Curry and Feys, 1958]. The connections between Church’s lambda calculi and programming languages, the applicative/imperative distinction, and the **let** notation are from Landin [1964; 1966]. The quantification notation is from [Tennent, 1987b]. Our treatment of substitution is based on the approach used in [Ebbinghaus *et al.*, 1984; Oles, 1987; Stoughton, 1988b]. Completeness of the logic is treated in [Friedman, 1975].

## 4 Recursion

Most programming languages allow *recursive* definitions, in which the name of the entity being defined can ‘recur’ in its own definition. In this section we introduce recursive variants of the **let** forms of definition discussed in Section 3, and describe the semantic techniques necessary to interpret recursion.

### 4.1 Recursive definitions

Consider the following syntax rule:

*Recursive function definition:*

$$\frac{\begin{array}{c} \left[ \begin{array}{c} \iota: \theta_0 \rightarrow \theta \\ \iota_0: \theta_0 \\ \vdots \\ P: \theta \end{array} \right] \quad \begin{array}{c} [\iota: \theta_0 \rightarrow \theta] \\ \vdots \\ Q: \theta' \end{array} \\ \text{letrec } \iota(\iota_0: \theta_0): \theta = P \text{ in } Q: \theta' \end{array}$$

Note that free occurrences of  $\iota$  in  $P$  are bound in the definition; that is, they are *recursive* uses of that function name (unless  $\iota_0 = \iota$ ). The result type  $\theta$  of the function must be explicitly indicated here; it cannot be straightforwardly inferred from  $P$  (because  $\iota$  will typically recur in  $P$ ).

We can unify and simplify our treatment of recursion by also introducing the following construct, which, in principle, allows for recursive definition of identifiers of arbitrary phrase types, rather than just functional types:

*Recursive definition:*

$$\frac{\begin{array}{c} [\iota: \theta] \\ \vdots \\ P: \theta \end{array} \quad \begin{array}{c} [\iota: \theta] \\ \vdots \\ Q: \theta' \end{array}}{\text{letrec } \iota: \theta \text{ be } P \text{ in } Q: \theta'}$$

The type of  $\iota$  must be explicitly indicated. We can now *define* recursive function definitions by the following equivalence:

$$\begin{aligned} \text{letrec } \iota(\iota_0: \theta_0): \theta = P \text{ in } Q \\ \equiv \text{letrec } \iota: \theta_0 \rightarrow \theta \text{ be } \lambda \iota_0: \theta_0. P \text{ in } Q. \end{aligned}$$

For example,

```
letrec double: val[nat] → val[nat] be
  λn: val[nat]. if n = 0 then 0 else 2 + double(n - 1)
in ...
```

defines the function  $\text{double}(n) = 2 \times n$ .

Programming languages often allow mutually recursive (simultaneous) definitions of *several* identifiers, as in

**letrec**  $\iota_1: \theta_1$  **be**  $P_1, \iota_2: \theta_2$  **be**  $P_2, \dots, \iota_n: \theta_n$  **be**  $P_n$  **in**  $Q$

Any free occurrence of identifiers  $\iota_1, \iota_2, \dots, \iota_n$  in phrases  $P_1, P_2, \dots, P_n$  or  $Q$  is bound by the recursive definition. This can also be treated as defined notation by repeatedly using the following transformation until only simple recursive definitions remain:

$$\begin{aligned} \text{letrec } \iota_1: \theta_1 \text{ be } P_1, \iota_2: \theta_2 \text{ be } P_2, \dots, \iota_n: \theta_n \text{ be } P_n \text{ in } Q \\ \equiv \text{letrec } \iota_1: \theta_1 \text{ be } (\text{letrec } \iota_2: \theta_2 \text{ be } P_2, \dots, \iota_n: \theta_n \text{ be } P_n \text{ in } P_1) \\ \text{in } (\text{letrec } \iota_2: \theta_2 \text{ be } P_2, \dots, \iota_n: \theta_n \text{ be } P_n \text{ in } Q) \end{aligned}$$

The equivalence shows how to ‘factor out’ the definition of  $\iota_1$  by using *separate* recursive definitions of  $\iota_2, \dots, \iota_n$  for  $P_1$  and  $Q$ .



The key idea in the semantic interpretation of (simple) recursive definitions is that the meaning defined for  $\iota$  by **letrec**  $\iota: \theta$  **be**  $P$  **in**  $\dots$  is the appropriate solution of the equation

$$p = \llbracket P \rrbracket(u \mid \iota \mapsto p).$$

Consider the definition of *double* above; if we substitute  $2 \times (n - 1)$  for *double*( $n - 1$ ) in the lambda expression we get

$$\begin{aligned} \lambda n: \text{val}[\text{nat}]. \text{if } n = 0 \text{ then } 0 \text{ else } 2 + (2 \times (n - 1)) \\ \equiv \lambda n: \text{val}[\text{nat}]. \text{if } n = 0 \text{ then } 0 \text{ else } 2 + (2 \times n) - 2 \\ \equiv \lambda n: \text{val}[\text{nat}]. 2 \times n \end{aligned}$$

and so the corresponding equation is indeed *satisfied* by the function defined by *double*( $n$ ) =  $2 \times n$ .

It is also the case that  $n = 0$  is the unique solution in  $N$  of the equation corresponding to the recursive definition

$$\text{letrec } n: \text{val}[\text{nat}] \text{ be } n \times 0 \text{ in } \dots;$$

but it is less evident that this will be a satisfactory way of dealing with recursive definitions such as

$$\text{letrec } n: \text{val}[\text{nat}] \text{ be } n + 1 \text{ in } \dots,$$

the corresponding equation to which is not satisfied by *any* natural number, or

$$\text{letrec } n: \text{val}[\text{nat}] \text{ be } n \text{ in } \dots,$$

the corresponding equation to which is satisfied by *every* natural number! But, operationally, such recursive definitions lead to *non-terminating* computations; when we have allowed for this possibility in our semantic model, we will have a way out of these difficulties.

In mathematics, an equation of the form

$$x = \dots x \dots$$

is termed a *fixed-point* equation (because a solution is preserved by substituting it into the right-hand side; i.e., it is a 'fixed point' for the right-hand side). Our intention is to 'solve' such equations in much the same way that we solved the equation

$$\llbracket \text{while } B \text{ do } C \rrbracket = \llbracket \text{if } B \text{ then } C ; \text{while } B \text{ do } C \rrbracket$$

in Section 2.3; i.e., by taking the 'limit' of a sequence of approximations obtained by starting from a trivial initial approximation and generating

better-and-better approximations by substituting for the indeterminate of the equation on the right-hand side. What are needed are generalizations of

- the meaning of **diverge**, which is the initial approximation appropriate to command meanings;
- the  $\sqsubseteq$  relation on partial-function graphs, which is the notion of approximation appropriate to command meanings; and
- the ‘limit’ obtained by forming the *union* of the approximating partial functions.

In the following section, we present enough of a theory of computation called *domain theory* to allow us to deal with recursive definitions in this way.

## 4.2 Domain-theoretic semantics

The existence and uniqueness of appropriate solutions to fixed-point equations are assured by adopting the following principles of domain theory.

1. A computational domain (hereafter abbreviated to just *domain*) is a partially-ordered set, where the partial order (usual notation:  $\sqsubseteq$ ) models approximation with respect to information content; furthermore, a domain is  $\omega$ -complete; i.e., it contains a least upper bound  $\bigsqcup_{i \in \omega} d_i$  of every  $\omega$ -chain  $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$ . We do not insist that every domain have a least element.
2. A computable function between domains preserves domain structure; i.e., it is monotonic (order-preserving) and continuous (limit-preserving).

The relevance of these principles to recursive definitions is established by the following theorem.

**Theorem 4.2.1 (Least Fixed Points).** *Let  $D$  be any domain with a least element  $\perp$ , and  $f: D \rightarrow D$  be any continuous function on  $D$ ; then  $\bigsqcup_{i \in \omega} f^i(\perp)$  is the least fixed point of  $f$ .*

**Proof.** It is easily proved by mathematical induction on  $i$  that, for every  $i \in \omega$ ,  $f^i(\perp) \sqsubseteq f^{i+1}(\perp)$  so that the limit is well-defined. To see that it is a fixed point of  $f$ :

$$\begin{aligned}
f(\bigsqcup_{i \in \omega} f^i(\perp)) &= \bigsqcup_{i \in \omega} f(f^i(\perp)) && (f \text{ is continuous}) \\
&= \bigsqcup_{i \in \omega} f^{i+1}(\perp) \\
&= \perp \sqcup \bigsqcup_{i \in \omega} f^{i+1}(\perp) && (\perp \sqcup d = d) \\
&= f^0(\perp) \sqcup \bigsqcup_{i \in \omega} f^{i+1}(\perp) && (f^0(\perp) = \perp) \\
&= \bigsqcup_{i \in \omega} f^i(\perp).
\end{aligned}$$

To see that  $\bigsqcup_{i \in \omega} f^i(\perp)$  is the *least* fixed point, consider any fixed point  $d'$  of  $f$ ; we first show by mathematical induction on  $i$  that, for every  $i \in \omega$ ,  $f^i(\perp) \sqsubseteq d'$ :  $f^0(\perp) = \perp \sqsubseteq d'$  and, if  $f^i(\perp) \sqsubseteq d'$  then  $f^{i+1}(\perp) \sqsubseteq f(d')$  (by monotonicity of  $f$ ) and  $f(d') = d'$  (by assumption). Hence,  $d'$  is an upper bound of  $\{f^i(\perp) \mid i \in \omega\}$  and so  $\bigsqcup_{i \in \omega} f^i(\perp) \sqsubseteq d'$ . ■

Before we can use this result to interpret recursive definitions, we must show that, for every computational phrase type  $\theta$  and phrase-type assignment  $\pi$ , we can define *domains*  $\llbracket \theta \rrbracket$  and  $\llbracket \pi \rrbracket$  with least elements  $\perp_\theta$  and  $\perp_\pi$ , respectively, and that, for every phrase  $X$ , we can interpret  $X$  as a *continuous* function  $\llbracket X \rrbracket$  from environments to meanings. For any domains  $D$  and  $E$ , define

- $D \times E$  to be the Cartesian product of the underlying sets, ordered component-wise; i.e.,  $(d, e) \sqsubseteq (d', e')$  iff  $d \sqsubseteq d'$  and  $e \sqsubseteq e'$ ;
- $D \rightarrow E$  to be the set of all continuous functions from  $D$  to  $E$ , ordered pointwise; i.e.,  $f \sqsubseteq g$  iff, for all  $d \in D$ ,  $f(d) \sqsubseteq g(d)$ ; and
- $D \rightharpoonup E$  to be the set of all continuous partial functions from  $D$  to  $E$ , ordered pointwise, where a partial function  $f$  is continuous iff it is monotonic (i.e., if  $f(d)$  is defined and  $d \sqsubseteq d'$  then  $f(d')$  is defined and  $f(d) \sqsubseteq f(d')$ ) and, for every  $\omega$ -chain  $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$ , if  $f(\bigsqcup_{i \in \omega} d_i)$  is defined then for some  $j \in \omega$ ,  $f(d_j)$  is defined and

$$f\left(\bigsqcup_{i \in \omega} d_i\right) = \bigsqcup_{i \in \omega} f(d_{i+j}).$$

Notice that  $f(d_i)$  might be undefined for  $i < j$ .

Finally, if  $D_i$  is a domain for every  $i \in I$ , let  $\prod_{i \in I} D_i$  be the set of all functions  $f$  from  $I$  to the union of the  $D_i$  such that, for every  $i \in I$ ,  $f(i) \in D_i$ , ordered point-wise. It can be verified that all of these are domains.

We can now re-interpret the phrase types of our applicative language as domains. We could regard all of the sets used previously as domains by assuming that they are *discretely ordered*; i.e.,  $d \sqsubseteq d'$  iff  $d = d'$ ; but the

resulting domains would not have least elements. If we want to allow use of recursively defined functions in expressions, the semantic model must allow for non-termination of expression evaluation. Hence, we re-define  $\llbracket \text{val}[\tau] \rrbracket$  to be the ‘lifted’ domain  $\llbracket \tau \rrbracket_\perp$  obtained by adding a new element  $\perp$  to the set  $\llbracket \tau \rrbracket$  and defining the partial order by  $d \sqsubseteq d'$  iff  $d = \perp$  or  $d = d'$ . We can continue to write

$$\llbracket \theta \rightarrow \theta' \rrbracket = \llbracket \theta \rrbracket \rightarrow \llbracket \theta' \rrbracket$$

and

$$\llbracket \pi \rrbracket = \prod_{\iota \in \text{dom } \pi} \llbracket \pi(\iota) \rrbracket,$$

but these are now interpreted as definitions of *domains* with non-trivial orderings; for example, any  $f \in \llbracket \theta \rightarrow \theta' \rrbracket$  must now be a *continuous* function, preserving limits of chains in  $\llbracket \theta \rrbracket$ . Every  $\llbracket \theta \rightarrow \theta' \rrbracket$  and  $\llbracket \pi \rrbracket$  now has a least element  $\perp_{\theta \rightarrow \theta'}$  or  $\perp_\pi$ , given inductively by  $\perp_{\theta \rightarrow \theta'}(a) = \perp_{\theta'}$  for all  $a \in \llbracket \theta \rrbracket$ , and  $\perp_\pi(\iota) = \perp_{\pi(\iota)}$  for all  $\iota \in \text{dom } \pi$ , respectively.

We now re-interpret the phrases of our applicative language by defining valuation functions

$$\llbracket \cdot \rrbracket_{\pi\theta} : [\theta]_\pi \rightarrow (\llbracket \pi \rrbracket \rightarrow \llbracket \theta \rrbracket)$$

so that every phrase  $X \in [\theta]_\pi$  is mapped into a *continuous* function from domain  $\llbracket \pi \rrbracket$  to domain  $\llbracket \theta \rrbracket$ . It is much simpler to describe the changes necessary than to present all of the semantic equations again.

- Each of the primitive operations (**not**, **<**, **=**, **succ**, **and**, ...) must be interpreted by a monotonic extension of the corresponding function on the data types (such as the ‘strict’ extension that produces  $\perp$  if any argument is  $\perp$ ).
- The conditional expression is interpreted as follows:

$$\llbracket \text{if } B \text{ then } X_0 \text{ else } X_1 \rrbracket_{\theta}(u) = \begin{cases} \llbracket X_0 \rrbracket_{\theta}(u), & \text{if } \llbracket B \rrbracket u = \text{true} \\ \llbracket X_1 \rrbracket_{\theta}(u), & \text{if } \llbracket B \rrbracket u = \text{false} \\ \perp_{\theta}, & \text{if } \llbracket B \rrbracket u = \perp \end{cases}$$

Note that, when the value of  $B$  is  $\perp$ , the result of the conditional is also  $\perp$ , even if  $\llbracket X_0 \rrbracket u = \llbracket X_1 \rrbracket u$ .

No other changes are needed. The following ensures that the functions on domains are in fact continuous.

**Proposition 4.2.2.** *For all  $X \in [\theta]_\pi$ ,  $\llbracket X \rrbracket_{\pi\theta}$  is continuous, and, when  $\theta = \theta' \rightarrow \theta''$ ,  $\llbracket X \rrbracket_{\pi(\theta' \rightarrow \theta'')}(u)$  is continuous for all  $u \in \llbracket \pi \rrbracket$ .*

The proof, by structural induction, is omitted.

Finally, we can treat the recursive-definition form,

$$(\text{letrec } \iota: \theta \text{ be } P \text{ in } Q) \in [\theta']_\pi.$$

Consider any  $u \in \llbracket \theta \rrbracket$  and define  $f: \llbracket \theta \rrbracket \rightarrow \llbracket \theta \rrbracket$  as follows:

$$f(p) = \llbracket P \rrbracket(u \mid \iota \mapsto p)$$

for every  $p \in \llbracket \theta \rrbracket$ ;  $f$  is continuous if  $\llbracket P \rrbracket$  is, and so we define

$$\llbracket \text{letrec } \iota: \theta \text{ be } P \text{ in } Q \rrbracket u = \llbracket Q \rrbracket \left( u \mid \iota \mapsto \bigsqcup_{i \in \omega} f^i(\perp_\theta) \right)$$

From Theorem 4.2.1 we know that the limit is well-defined, and that the meaning defined for  $\iota$  is the least solution of the equation

$$p = \llbracket P \rrbracket(u \mid \iota \mapsto p).$$

For example, consider evaluating

**letrec**  $g: \text{val}[\text{nat}] \rightarrow \text{val}[\text{nat}]$  **be**  
 $\lambda n: \text{val}[\text{nat}]. \text{if } n = 0 \text{ then } 0 \text{ else } 2 + g(n - 1)$   
**in**  $\dots$ ;

in environment  $u$ ; then

$$f(p) = \llbracket \lambda n: \text{val}[\text{nat}]. \text{if } n = 0 \text{ then } 0 \text{ else } 2 + g(n - 1) \rrbracket (u \mid g \mapsto p),$$

for all  $p \in \llbracket \text{val}[\text{nat}] \rightarrow \text{val}[\text{nat}] \rrbracket$ , so that

$$f(p)(a) = \begin{cases} 0, & \text{if } a = 0 \\ 2 + p(a - 1), & \text{if } a > 0 \\ \perp, & \text{if } a = \perp \end{cases}$$

for all  $a \in \llbracket \text{val}[\text{nat}] \rrbracket$ . The approximations to the least fixed point are the functions defined as follows:

$$\begin{aligned} p_0(a) &= \perp \\ p_1(a) &= \begin{cases} 0, & \text{if } a = 0 \\ 2 + p_0(a - 1), & \text{if } a > 0 \\ \perp, & \text{if } a = \perp \end{cases} \\ &= \begin{cases} 0, & \text{if } a = 0 \\ \perp, & \text{otherwise;} \end{cases} \end{aligned}$$



$$\begin{aligned}
p_2(a) &= \begin{cases} 0, & \text{if } a = 0 \\ 2 + p_1(a - 1), & \text{if } a > 0 \\ \perp, & \text{if } a = \perp \end{cases} \\
&= \begin{cases} 0, & \text{if } a = 0 \\ 2, & \text{if } a = 1 \\ \perp, & \text{otherwise;} \end{cases} \\
p_3(a) &= \begin{cases} 0, & \text{if } a = 0 \\ 2 + p_2(a - 1), & \text{if } a > 0 \\ \perp, & \text{if } a = \perp \end{cases} \\
&= \begin{cases} 0, & \text{if } a = 0 \\ 2, & \text{if } a = 1 \\ 4, & \text{if } a = 2 \\ \perp, & \text{otherwise;} \end{cases}
\end{aligned}$$

and so on; i.e., for any  $i \in \omega$ ,

$$p_i(a) = \begin{cases} 2 \times a, & \text{if } a < i \\ \perp, & \text{otherwise,} \end{cases}$$

and so their least upper bound is the function  $p_\infty$  defined by

$$p_\infty(a) = \begin{cases} 2 \times a, & \text{if } a \neq \perp \\ \perp, & \text{if } a = \perp. \end{cases}$$

We must verify that  $\llbracket \text{letrec } \iota: \theta \text{ be } P \text{ in } Q \rrbracket$  is continuous. We can simplify the problem by introducing a new constant

$$\text{rec}[\theta]: (\theta \rightarrow \theta) \rightarrow \theta$$

for every phrase type  $\theta$  in our programming language, defining the **letrec** form by the following equivalence:

$$\begin{aligned}
\text{letrec } \iota: \theta \text{ be } P \text{ in } Q &\equiv \text{let } \iota \text{ be } \# \text{rec}[\theta] \iota. P \text{ in } Q \\
&\equiv \text{let } \iota \text{ be } \text{rec}[\theta](\lambda \iota: \theta. P) \text{ in } Q,
\end{aligned}$$

and defining

$$\llbracket \text{rec}[\theta] \rrbracket u f = \bigsqcup_{i \in \omega} f^i(\perp_\theta)$$

for every environment  $u$  and  $f \in \llbracket \theta \rrbracket \rightarrow \llbracket \theta \rrbracket$ . We now only need to show that  $\llbracket \text{rec}[\theta] \rrbracket$  is continuous.

**Proposition 4.2.3.** *If  $f \in D \rightarrow D$  is continuous and  $D$  has a least element  $\perp$ , the function  $\text{fix}(f) = \bigsqcup_{i \in \omega} f^i(\perp)$  is continuous.*

**Proof.** Let  $fix_i(f) = f^i(\perp)$  for every  $i \in \omega$ ; then  $fix(f) = \bigsqcup_{i \in \omega} f^i(\perp) = \bigsqcup_{i \in \omega} fix_i(f) = (\bigsqcup_{i \in \omega} fix_i)(f)$  by the definition of limits in domains of functions, so that  $fix = \bigsqcup_{i \in \omega} fix_i$  and because the  $fix_i$  are easily shown to be continuous, so is the limit by the fact that the least upper bound of a chain of continuous functions is continuous. ■

Hence the interpretations of the constants  $\mathbf{rec}[\theta]$  are continuous functions, and so, by Proposition 4.2.2, the interpretation of the **letrec** definition is continuous.

Defining recursive definitions in terms of **rec** also makes it clear that the Coherence and Substitution lemmas remain valid if recursion is added to the applicative language. Furthermore, the equivalence

$$\mathbf{let} \ \iota \ \mathbf{be} \ P \ \mathbf{in} \ Q \equiv [Q](\iota \mapsto P)$$

remains valid, and this has an important consequence for implementations: a ‘call-by-name’ or ‘lazy’ implementation of parameter passing is called for. For example,

$$\mathbf{let} \ n \ \mathbf{be} \ N \ \mathbf{in} \ \mathbf{true}$$

and

$$(\lambda n: \theta. \mathbf{true})(N)$$

are both equivalent to **true** for every  $N: \theta$ , even if an evaluation of  $N$  would fail to terminate. A (sequential) implementation must therefore defer evaluations of actual-parameter expressions like  $N$  until (if ever) the value of that expression is actually needed.

We can make explicit the parallel between the semantic valuation of **letrec**  $\iota: \theta \ \mathbf{be} \ P \ \mathbf{in} \ Q$  and the interpretation of the **while** loop in Section 2.3 by using the Substitution lemma. Let  $\mathbf{undef}[\theta]: \theta \ \mathbf{be} \ \text{a constant}$  for every phrase type  $\theta$  such that  $\llbracket \mathbf{undef}[\theta] \rrbracket u = \perp_\theta$  for every environment  $u$ . We can then define a sequence  $P_i$  for  $i \in \omega$  of phrases of type  $\theta$  as follows:

$$\begin{aligned} P_0 &= \mathbf{undef}[\theta] \\ P_{i+1} &= [P](\iota \mapsto P_i) \end{aligned}$$

For any appropriate environment  $u$ , we can then define the corresponding sequence  $p_i = \llbracket P_i \rrbracket u$  of elements of  $\llbracket \theta \rrbracket$  as follows:

$$\begin{aligned} p_0 &= \llbracket P_0 \rrbracket u = \perp_\theta \\ p_{i+1} &= \llbracket P_{i+1} \rrbracket u = \llbracket [P](\iota \mapsto P_i) \rrbracket u \\ &= \llbracket P \rrbracket (u \mid \iota \mapsto \llbracket P_i \rrbracket u) \quad (\text{Substitution lemma}) \\ &= \llbracket P \rrbracket (u \mid \iota \mapsto p_i) \end{aligned}$$

It is clear that  $p_i \sqsubseteq p_{i+1}$  for every  $i \in \omega$  and hence the limit  $p_\infty = \bigsqcup_{i \in \omega} p_i$  exists; then

$$\llbracket \text{letrec } \iota: \theta \text{ be } P \text{ in } Q \rrbracket u = \llbracket Q \rrbracket (u \mid \iota \mapsto p_\infty).$$

Clearly, if  $f(p) = \llbracket P \rrbracket (u \mid \iota \mapsto p)$ ,  $p_i = f^i(\perp_\theta)$  for every  $i \in \omega$ , and  $p_\infty = \bigsqcup_{i \in \omega} f^i(\perp_\theta)$ .

We conclude this section with a result that will justify an important method of reasoning about meanings that are recursively defined.

**Theorem 4.2.4.** *Let  $D$  be a domain with least element  $\perp$ ,  $\varphi$  be a predicate on  $D$  such that, for any  $d \in D^\omega$ ,  $\varphi(d_i)$  for all  $i \in \omega$  implies  $\varphi(\bigsqcup_{i \in \omega} d_i)$ , and  $f: D \rightarrow D$  be a continuous function; if  $\varphi(\perp)$  and, for all  $d \in D$ ,  $\varphi(d)$  implies  $\varphi(f(d))$ , then  $\varphi(\text{fix}(f))$ .*

**Proof.** It can be shown by mathematical induction on  $i$  that  $\varphi(f^i(\perp))$  for all  $i \in \omega$ ; then the limit-completeness of the subset of  $D$  satisfying  $\varphi$  allows us to conclude that  $\varphi(\bigsqcup_{i \in \omega} f^i(\perp))$ ; i.e.,  $\varphi(\text{fix}(f))$ . ■

This result justifies the following inductive method of reasoning known as *fixed-point* (or *computational*) induction: a predicate on a domain with a least element  $\perp$  is proved

- (i) for  $\perp$  (the basis), and
- (ii) for  $f(d)$ , on the assumption (the inductive hypothesis) that  $d$  has the property.

The conclusion is that  $\text{fix}(f)$  has the property, provided that the property is ‘admissible for fixed-point induction’, i.e., satisfies the limit-completeness requirement of the theorem. This is a difficult issue in general; we address it for a specific formal language of predicates in Section 4.4.

### 4.3 Operational semantics

In this section, we describe an operational semantics for our applicative language, and verify its correctness using the denotational semantics. The operational semantics will be expressed as a family of binary relations  $\succ_\theta$  on  $[\theta]_{\pi_0}$  for every phrase type  $\theta$ , where  $\pi_0$  is the phrase-type assignment for pre-defined identifiers. For  $P, P' \in [\theta]_{\pi_0}$ , the relationship  $P \succ_\theta P'$  should hold just when the change from  $P$  to  $P'$  is a single computational step.

The  $\succ_\theta$  relations can be defined to be the smallest binary relations satisfying the axioms and rules of Table 9 and similar ones for the other operators of the language and for the pre-defined identifiers.

$$\begin{array}{c} \overline{(P) \succ_\theta P} \\[10pt] \overline{(\lambda \iota: \theta. P)Q \succ_{\theta'} [P](\iota \mapsto Q)} \\[10pt] \overline{\text{rec}[\theta](F) \succ_\theta F(\text{rec}[\theta](F))} \end{array}$$

$$\begin{array}{c}
\frac{F \succ_{\theta \rightarrow \theta'} F'}{F(A) \succ_{\theta'} F'(A)} \\
\\
\frac{}{\text{if true then } P_1 \text{ else } P_2 \succ_{\theta} P_1} \\
\\
\frac{}{\text{if not true then } P_1 \text{ else } P_2 \succ_{\theta} P_2} \\
\\
\frac{B \succ B'}{\text{if } B \text{ then } P_1 \text{ else } P_2 \succ \text{if } B' \text{ then } P_1 \text{ else } P_2} \\
\\
\frac{}{\text{undef}[\theta] \succ_{\theta} \text{undef}[\theta]} \\
\\
\frac{}{\text{not not true} \succ_{\text{val}[\text{bool}]} \text{true}} \\
\\
\frac{B \succ_{\text{val}[\text{bool}]} B'}{\text{not } B \succ_{\text{val}[\text{bool}]} \text{not } B'}
\end{array}$$

Table 9. Operational semantics of expressions

Provided that the  $\succ_{\theta}^*$  relations are functional for every  $\theta$  of the form  $\text{val}[\tau]$ , we can define valuations

$$\{\cdot\}_{\tau}: [\text{val}[\tau]]_{\pi_0} \rightarrow \llbracket \text{val}[\tau] \rrbracket$$

on programs as follows:

$$\{\!|P|\!\}_{\tau} = \begin{cases} \llbracket P' \rrbracket u_0, & \text{if } P \succ_{\text{val}[\tau]}^* P' \text{ and } P' \not\succ_{\text{val}[\tau]} P'' \text{ for any } P'', \\ \perp, & \text{otherwise.} \end{cases}$$

The denotational valuation  $\llbracket \cdot \rrbracket$  is used here only to map ‘canonical’ (i.e., irreducible) programs (such as **true** and **not true**) to their values.

We want now to prove that the operational and denotational valuations for programs are equivalent in the sense that  $\{\!|P|\!\}_{\tau} = \llbracket P \rrbracket_{\pi_0 \text{val}[\tau]}(u_0)$  for all programs  $P \in [\text{val}[\tau]]_{\pi_0}$ . We can prove that  $\{\!|P|\!\}_{\tau} \subseteq \llbracket P \rrbracket_{\pi_0 \text{val}[\tau]}(u_0)$  by showing that for each axiom in Table 9,

$$\llbracket P \rrbracket u_0 = \llbracket P' \rrbracket u_0 \text{ when } P \succ_{\theta} P'$$

and that this property is preserved by each of the rules, and so, by induction on the number of computation steps, if  $P \succ_{\theta}^* P'$ , then  $\{\!|P|\!\}_{\tau} = \llbracket P' \rrbracket u_0 = \llbracket P \rrbracket u_0$ .

In the other direction, we show that  $\llbracket P \rrbracket_{\pi_0 \text{val}[\tau]}(u_0) \subseteq \{\!|P|\!\}_{\tau}$  by proving a more general property by structural induction. We first define a family

of binary relations  $\|\theta\| \subseteq \llbracket \theta \rrbracket \times [\theta]_{\pi_0}$  by induction on the structure of  $\theta$  as follows:

- $v \|\mathbf{val}[\tau]\| V$  if and only if  $v \sqsubseteq \{V\}_\tau$ ;
- $f \|\theta \rightarrow \theta'\| F$  if and only if, for all  $a \in \llbracket \theta \rrbracket$  and  $A \in [\theta]_{\pi_0}$ ,  $a \|\theta\| A$  implies  $f(a) \|\theta'\| F(A)$ .

Such families of relations, defined by induction on types, are termed *logical relations*. Intuitively,  $p \|\theta\| P$  holds when  $p \in \llbracket \theta \rrbracket$  approximates (in the sense of  $\sqsubseteq$ ) the ‘operational meaning’ of  $P$ .

**Proposition 4.3.1.** *For all  $u \in \llbracket \pi \rrbracket$ ,  $\sigma \in [\pi]_{\pi_0}$  and  $X \in [\theta]_\pi$ , if*

$$u(\iota) \|\pi(\iota)\| \sigma(\iota)$$

*for every  $\iota \in \text{dom } \pi$  then*

$$\llbracket X \rrbracket_{\pi\theta}(u) \|\theta\| [X](\sigma).$$

**Proof.** The proof is by structural induction over  $X$ . We discuss only the case of  $\mathbf{rec}[\theta]$  when  $\theta = \mathbf{val}[\tau]$ .

Consider any  $f \in \llbracket \theta \rrbracket \rightarrow \llbracket \theta \rrbracket$  and  $F \in [\theta \rightarrow \theta]_{\pi_0}$  such that  $f \|\theta \rightarrow \theta\| F$ , and let  $m = \{\mathbf{rec}[\theta](F)\}_\tau$ ; then we show that  $\text{fix}(f) \sqsubseteq m$  by fixed-point induction using  $\varphi(d) = (d \sqsubseteq m)$ , which is obviously admissible. The basis is clear. Assume  $\varphi(d)$ ; i.e.,  $d \sqsubseteq m = \{\mathbf{rec}[\theta](F)\}_\tau$  (because  $\theta = \mathbf{val}[\tau]$ ), and then  $\varphi(f(d))$  follows from the assumption that  $f \|\theta \rightarrow \theta\| F$  and the fact that  $\mathbf{rec}[\theta](F) \succ_\theta F(\mathbf{rec}[\theta](F))$ . ■

In particular,  $\llbracket P \rrbracket_{\pi_0 \mathbf{val}[\tau]}(u_0) \sqsubseteq \{P\}_\tau$  for every program  $P$ , provided  $u_0(\iota) \|\pi_0(\iota)\| \iota$  for every  $\iota \in \text{dom } \pi_0$  (i.e., the meanings of the pre-defined identifiers are correctly implemented), and so we can conclude that the operational and denotational valuations for programs are equivalent.

**Exercise 4.3.2.** Where would the proof of Proposition 4.3.1 break down if  $v \|\mathbf{val}[\tau]\| V$  were defined to be  $v = \{V\}_\tau$ ? (This would permit the equivalence of the denotational and operational valuations to be shown with a single induction.)

## 4.4 Programming logic

In this section we discuss a formal system based on LCF (‘Logic for Computable Functions’) for reasoning about programs in our applicative language (augmented by recursion). It is similar to the system described in Section 3.5.



### 4.4.1 Syntax and semantics

The language of specifications will be identical to that of Section 3.5, except that we take *approximation* formulas as the atomic specifications:

*Approximation:*

$$\frac{X_0: \theta \quad X_1: \theta}{X_0 \sqsubseteq_{\theta} X_1: \text{spec}}$$

with

$$\llbracket X_0 \sqsubseteq_{\theta} X_1 \rrbracket u = (\llbracket X_0 \rrbracket u \sqsubseteq \llbracket X_1 \rrbracket u),$$

and then define  $X_0 \equiv_{\theta} X_1$  to be an abbreviation for

$$X_0 \sqsubseteq_{\theta} X_1 \ \& \ X_1 \sqsubseteq_{\theta} X_0.$$

### 4.4.2 Axioms

The axioms of the formal system are as follows: the Substitutivity, Alpha, Beta, and Eta laws of Section 3.5 and the following:

*Reflexivity:*

$$\pi \vdash X \sqsubseteq_{\theta} X,$$

when  $X \in [\theta]_{\pi}$ .

*Transitivity:*

$$\pi \vdash X_0 \sqsubseteq_{\theta} X_1 \ \& \ X_1 \sqsubseteq_{\theta} X_2 \Rightarrow X_0 \sqsubseteq_{\theta} X_2,$$

when  $X_0, X_1, X_2 \in [\theta]_{\pi}$ .

*Monotonicity:*

$$\pi \vdash X_0 \sqsubseteq_{\theta} X_1 \Rightarrow F(X_0) \sqsubseteq_{\theta'} F(X_1),$$

when  $X_0, X_1 \in [\theta]_{\pi}$  and  $F \in [\theta \rightarrow \theta']_{\pi}$ .

*Extensionality:*

$$\pi \vdash (\forall \iota: \theta. F_0(\iota) \sqsubseteq_{\theta'} F_1(\iota)) \iff F_0 \sqsubseteq_{\theta \rightarrow \theta'} F_1,$$

when  $F_0, F_1 \in [\theta \rightarrow \theta']_{\pi}$  and  $\iota \notin \text{dom } \pi$ .

*Undefined:*

$$\pi \vdash \text{undef}[\theta] \sqsubseteq_{\theta} X,$$

when  $X \in [\theta]_{\pi}$ .

*Fixed point:*

$$\pi \vdash \mathbf{rec}[\theta](F) \equiv_{\theta} F(\mathbf{rec}[\theta](F)),$$

when  $F \in [\theta \rightarrow \theta]_{\pi}$ .

*Fixed-point induction:*

$$\begin{aligned} \pi \vdash [Z](\iota \mapsto \mathbf{undef}[\theta]) \ \& \ (\forall \iota: \theta. Z \Rightarrow [Z](\iota \mapsto F(\iota))) \\ \Rightarrow [Z](\iota \mapsto \mathbf{rec}[\theta](F)), \end{aligned}$$

when  $Z \in [\mathbf{spec}]_{(\pi|\iota \mapsto \theta)}$  and  $F \in [\theta \rightarrow \theta]_{\pi}$ , provided that  $\iota \notin \text{dom } \pi$  and, to ensure admissibility for fixed-point induction,  $\iota \notin \text{free}_{-}(Z)$ , where, for any  $Z \in [\mathbf{spec}]_{\pi}$ ,  $\text{free}_{-}(Z)$  and  $\text{free}_{+}(Z)$  are subsets of  $\text{free}(Z)$  simultaneously defined by induction on  $Z$  as follows:

$$\begin{aligned} \text{free}_{-}(X_0 \sqsubseteq_{\theta'} X_1) &= \emptyset \\ \text{free}_{-}(\mathbf{absurd}) &= \emptyset \\ \text{free}_{-}(Z_0 \ \& \ Z_1) &= \text{free}_{-}(Z_0) \cup \text{free}_{-}(Z_1) \\ \text{free}_{-}(Z_0 \Rightarrow Z_1) &= \text{free}_{+}(Z_0) \cup \text{free}_{-}(Z_1) \\ \text{free}_{-}(\forall \iota': \theta'. Z') &= \text{free}_{-}(Z') - \{\iota'\} \\ \\ \text{free}_{+}(X_0 \sqsubseteq_{\theta'} X_1) &= \text{free}(X_1) \\ \text{free}_{+}(\mathbf{absurd}) &= \emptyset \\ \text{free}_{+}(Z_0 \ \& \ Z_1) &= \text{free}_{+}(Z_0) \cup \text{free}_{+}(Z_1) \\ \text{free}_{+}(Z_0 \Rightarrow Z_1) &= \text{free}_{-}(Z_0) \cup \text{free}_{+}(Z_1) \\ \text{free}_{+}(\forall \iota': \theta'. Z') &= \text{free}_{+}(Z') - \{\iota'\} \end{aligned}$$

The negative sign in  $\text{free}_{-}$  refers to its use as a side condition in the Induction axiom, and the positive sign in  $\text{free}_{+}$  refers to its use on the ‘negative’ (left) argument of the implication in the definition of  $\text{free}_{-}(Z_0 \Rightarrow Z_1)$ . These definitions make the sets  $\text{free}_{-}$  and  $\text{free}_{+}$  as small as possible, while maintaining the following semantic properties.

**Proposition 4.4.1.** *Let  $Z \in [\mathbf{spec}]_{(\pi|\iota \mapsto \theta)}$ :*

- (a) *if  $\iota \notin \text{free}_{-}(Z)$  then, for all  $u \in \llbracket \pi \rrbracket$  and chains  $d \in \llbracket \theta \rrbracket^{\omega}$ , if, for all  $i \in \omega$ ,  $\llbracket Z \rrbracket(u \mid \iota \mapsto d_i)$ , then  $\llbracket Z \rrbracket(u \mid \iota \mapsto \bigsqcup_{i \in \omega} d_i)$ ; and*
- (b) *if  $\iota \notin \text{free}_{+}(Z)$  then, for all  $u \in \llbracket \pi \rrbracket$  and chains  $d \in \llbracket \theta \rrbracket^{\omega}$ , if  $\llbracket Z \rrbracket(u \mid \iota \mapsto \bigsqcup_{i \in \omega} d_i)$  then, for some  $i \in \omega$ ,  $\llbracket Z \rrbracket(u \mid \iota \mapsto d_i)$ .*

**Proof.** This can be proved by a simultaneous structural induction over  $Z$ . We will discuss only the case that  $\iota \notin \text{free}_{-}(Z_0 \Rightarrow Z_1)$ ; the proof is due to P. Cenciarelli.

Consider any  $u \in \llbracket \pi \rrbracket$  and chain  $d \in \llbracket \theta \rrbracket^{\omega}$ , and assume that, for all  $i \in \omega$ ,  $\llbracket Z_0 \rrbracket(u \mid \iota \mapsto d_i)$  implies  $\llbracket Z_1 \rrbracket(u \mid \iota \mapsto d_i)$ , and that  $\llbracket Z_0 \rrbracket(u \mid \iota \mapsto \bigsqcup_{i \in \omega} d_i)$ ; then  $\llbracket Z_0 \rrbracket(u \mid \iota \mapsto d_i)$  for some  $i \in \omega$ , by the induction hypothesis and the fact that  $i \notin \text{free}_{+}(Z_0)$ . But the subchain of  $d$  without  $d_i$  has the

same limit as  $d$ , and so we may conclude that there must be an *infinite* subchain of the  $d_i$  such that  $\llbracket Z_0 \rrbracket(u \mid \iota \mapsto d_i)$ , and having the same limit as  $d$ ; otherwise, we could prune the finite number of such  $d_i$  from the chain and obtain a subchain with the same limit that would contradict the induction hypothesis for  $Z_0$ . Then, by assumption,  $\llbracket Z_1 \rrbracket(u \mid \iota \mapsto d_i)$  for all  $d_i$  in the subchain, and hence  $\llbracket Z_1 \rrbracket(u \mid \iota \mapsto \bigsqcup_{i \in \omega} d_i)$  by the induction hypothesis on  $Z_1$  and the fact that  $\iota \notin \text{free}_-(Z_1)$ . ■

Validity of the fixed-point induction axiom follows from the Substitution lemma, Theorem 4.2.4 and the above proposition, which shows that  $\iota \notin \text{free}_-(Z)$  is a sufficient condition for admissibility for fixed-point induction of  $Z$  with respect to  $\iota$ .

The following is an example of a derivation in this system:

$$\begin{array}{c}
 \dfrac{\dfrac{[x: \theta] \quad [x \sqsubseteq_\theta A]}{F(x) \sqsubseteq_\theta F(A)} \text{ (Monot.)} \quad [F(A) \sqsubseteq_\theta A]}{F(x) \sqsubseteq_\theta A} \text{ (Trans.)} \\
 \dfrac{\dfrac{\text{undef}[\theta] \sqsubseteq_\theta A \text{ (Undef.)} \quad \dfrac{x \sqsubseteq_\theta A \Rightarrow F(x) \sqsubseteq_\theta A}{\forall x: \theta. x \sqsubseteq_\theta A \Rightarrow F(x) \sqsubseteq_\theta A} (\Rightarrow\text{-I})}{\text{rec}[\theta](F) \sqsubseteq_\theta A} (\forall\text{-I})}{F(A) \sqsubseteq_\theta A \Rightarrow \text{rec}[\theta](F) \sqsubseteq_\theta A} \text{ (Induct.)} \\
 \dfrac{\text{rec}[\theta](F) \sqsubseteq_\theta A}{F(A) \sqsubseteq_\theta A \Rightarrow \text{rec}[\theta](F) \sqsubseteq_\theta A} (\Rightarrow\text{-I})
 \end{array}$$

where  $\Rightarrow\text{-I}$  is the usual rule of Implication-Introduction; its use discharges an assumption by making it the precedent of the implication formula derived. This is a formal proof (with a few steps omitted) of what is known as Park's theorem [Park, 1969]:  $F(A) \sqsubseteq_\theta A \Rightarrow \text{rec}[\theta](F) \sqsubseteq_\theta A$ , when  $F \in [\theta \rightarrow \theta]_\pi$  and  $A \in [\theta]_\pi$ .

## 4.5 Full abstraction

We conclude our study of the semantics of the purely-applicative language by stating without proof some important results about full abstraction. It may be recalled from Section 1.4 that a semantic interpretation is said to be fully abstract if and only if phrase meanings are not distinguished unless there is a program context in which the phrases can be used that allows the distinction to be observed. It can be proved using domain-theoretic techniques beyond the scope of this chapter that the semantic interpretation of our language is in fact fully abstract, provided 'parallel conditionals' are added for types of the form  $\text{val}[\tau]$ ; i.e., **parif** conditionals that are like the conventional **if** conditionals except that

$$\text{parif undef}[\text{bool}] \text{ then } V \text{ else } V \equiv_{\text{val}[\tau]} V$$

for all  $V: \text{val}[\tau]$ .

```

F0 = λf: θ0. if f(true, undef[val[bool]]) then
                if f(undef[val[bool]], true) then
                    if f(false, false) then undef[val[nat]] else 0
                else undef[val[nat]]
            else undef[val[nat]]

```

$$F_0 \not\equiv_{\theta_0 \rightarrow \text{val}[\text{nat}]} F_1;$$
$$F_0(A) \equiv_{\text{val}[\text{nat}]} F_1(A)$$

Another approach to full abstraction is to define ‘smaller’ domains. For example, for the language we have considered here, we might want only ‘sequentially implementable’ functions, which would not include operations like the parallel *or*; however, despite considerable effort, the full-abstraction problem for this language has not yet been solved.

Some programming languages have type ‘loopholes’ or are deliberately designed to be *untyped* or allow the programmer to define types recursively. In such languages, it may even be possible for a procedure to be applicable to *itself*. For example, the following definition is legal in ALGOL 60:

```
integer procedure  $p(q, n)$ ;  
  integer procedure  $q$  ; integer  $n$ ;  
   $p :=$  if  $n = 0$  then 1 else  $n \times q(q, n - 1)$ 
```

A call of the form  $p(p, n)$  for  $n \geq 0$  returns the factorial of  $n$ . Note that the definition is *not* recursive; however, a domain of meanings appropriate for procedures such as  $p$  would need to satisfy the following domain equation:

$$D \cong D \times I \rightarrow I$$

where  $I$  is the domain of meanings of integer-valued expressions or value phrases. In our explicitly-typed language, an analogous procedure definition could not be written because it would be necessary to specify the *argument* type of parameter  $q$ .

As another example, the *untyped* lambda calculus is a language which resembles the *typed* lambda calculus (on which the simple applicative language of this Chapter is based), but has no type constraints. There is just a single phrase type, say **term**, and the syntax rules are as follows:

*Untyped application:*

$$\frac{P: \text{term} \quad Q: \text{term}}{PQ: \text{term}}$$

*Untyped abstraction:*

$$\frac{\begin{array}{c} [\iota: \text{term}] \\ \vdots \\ P: \text{term} \end{array}}{\lambda \iota. P: \text{term}}$$

Note that there is no need to specify the type of the bound identifier in the rule for Untyped abstraction (because there is only one type), and that the rule for Untyped application allows *any* term to be applied to *any* term (including itself).

For example, consider the term  $Y$ , as follows:

$$\lambda F. (\lambda x. F(xx)) (\lambda x. F(xx))$$

It is easily verified that  $Y$  has the property that, for *any* term  $F$ ,  $Y(F)$  is equivalent to  $F(Y(F))$  using the untyped analogue of the Beta law.  $Y$  is called a *fixed-point combinator*. Essentially, this shows that it is not necessary to *add* recursion to the *untyped* lambda calculus: recursion is already defineable in the pure system using  $Y$ .

A domain appropriate for this language would satisfy the isomorphism

$$D \cong D \rightarrow D$$

The non-trivial solution of this domain equation obtained by D. Scott in 1970 was the first natural model of the untyped lambda calculus. Other



applications of such domain equations will be discussed in Sections 6.4 and 6.7.

## 4.7 Bibliographic notes

The applicative language with recursion treated here is essentially what is termed PCF in [Plotkin, 1977]. The reduction of simultaneous recursion to iterated simple recursion was first presented in [Bekič, 1969]. The treatment of recursion in terms of fixed points of continuous functions in domain-like structures was first outlined in a privately circulated memo [Scott, 1969]; [Milner, 1973] is a detailed exposition of this work. Two of the best presentations of elementary domain theory are [Wadsworth, 1978] and [Plotkin, 1978], which originated as sets of lecture notes, but have been widely circulated. Other expositions may be found in many places, such as [Loeckx and Sieber, 1984; Schmidt, 1986; Gunter and Scott, 1990; Winskel, 1993] and the chapter on domain theory in this volume.

Various versions of the formal system called LCF have been described in [Scott, 1969; Milner, 1972; Milner *et al.*, 1975; Gordon *et al.*, 1979; Paulson, 1987]. Our treatment of admissibility for fixed-point induction is based on the approach used (for a different language of predicates) in [Reynolds, 1982]. The results on full abstraction are from [Plotkin, 1977]; other work on full abstraction may be found in [Milner, 1977; Berry *et al.*, 1985; Mulmuley, 1987; Stoughton, 1988a; Cartwright and Felleisen, 1992; Sieber, 1992]. Semantic models of the untyped lambda calculus are described in [Scott, 1970; Scott, 1972b; Wadsworth, 1976; Barendregt, (revised edition) 1984].

## 5 An Algol-like language I

In the preceding sections, we have analysed two programming languages:

- a simple *imperative* language with state-dependent expressions and state-changing commands, and
- a purely *applicative* language with state-independent expressions and functions, and recursion.

For each language we presented denotational, operational, and axiomatic semantics, and used the denotational description to verify the correctness of the operational and axiomatic ones.

The reader may be wondering whether the methods described so far are also applicable to the ‘procedural’ languages used in practice, which have both applicative and imperative aspects. In this section, we will see that this is definitely the case. The (syntax and semantics of the) two mini-languages will essentially be *combined* to form the ‘core’ of a procedural language. Remarkably, the combined language retains nearly all of the logical properties of the ‘pure’ languages and can reasonably be described as being *both imperative and applicative*.

In the following sections, we will add important new facilities such as jumps, local-variable declarations, data structures, and modules to this core. The real-life language that our language most closely resembles in semantic structure is ALGOL60 [Naur *et al.*, 1963], and so we term it an

*Algol-like* language. An alternative approach to combining imperative and applicative concepts will be discussed briefly for comparison purposes in Section 5.3, which can be skipped without loss of continuity.

## 5.1 Syntax

We begin by combining the type rules for the simple imperative and applicative languages as follows:

$$\theta ::= \mathbf{val}[\tau] \mid \mathbf{exp}[\tau] \mid \mathbf{comm} \mid \theta \rightarrow \theta' \mid (\theta)$$

where, as before,  $\tau$  ranges over data types. This combination gives us more than the *union* of the phrase types of those languages: we now have new *procedural* phrase types of the forms  $\theta \rightarrow \mathbf{exp}[\tau]$  and  $\theta \rightarrow \mathbf{comm}$ . Meanings of these types are like the ‘functions’ and procedures, respectively, of a language like PASCAL.

In the imperative language of Section 2, *all* identifiers were variable-identifiers, whereas in the applicative language of the preceding two sections, identifiers denoted values and functions but *not* variables. To allow here for variable-identifiers (and other kinds of variable-denoting phrases), we add new primitive phrase types of the form  $\mathbf{var}[\tau]$  for every data type  $\tau$ :

$$\theta ::= \dots \mid \mathbf{var}[\tau]$$

A variable-identifier for values of type  $\tau$  is now just an identifier having phrase type  $\mathbf{var}[\tau]$ .

This now also gives us procedural types of the forms  $\mathbf{var}[\tau] \rightarrow \theta$  and  $\theta \rightarrow \mathbf{var}[\tau]$ . The former are just procedures with variable parameters. Procedures with  $\mathbf{var}$  result types have been termed ‘selectors’; for example, conventional *arrays* (‘subscriptable variables’) are essentially procedures with an  $\mathbf{exp}$  argument type and a  $\mathbf{var}$  result type.

As the initial syntax of our language, we take *all* of the rules of the preceding three sections (including the ‘defined’ notations of Sections 3.3, 4.1 and 4.2), with a few small changes as follows.

The rules for variable-identifier expressions and assignment commands in Section 2.2 are no longer appropriate because we now have identifiers that do not denote variables. Also, we will want to have variable phrases that are not identifiers; for example, if  $\iota_1$  and  $\iota_2$  are variable-identifiers for values of type  $\tau$ ,  $E: \mathbf{exp}[\tau]$ , and  $B: \mathbf{exp}[\mathbf{bool}]$ , then one would expect the assignment command

$$(\mathbf{if } B \mathbf{ then } \iota_1 \mathbf{ else } \iota_2) := E,$$

which uses a ‘conditional variable’, to be equivalent to the following conditional command:

**if  $B$  then  $(\iota_1 := E)$  else  $(\iota_2 := E)$ .**

This illustrates that the left-hand sides of assignments can be more complex than simple identifiers. (In fact, few programming languages allow conditional variables, but the same kind of syntactic and semantic complexity is exhibited by many other features, such as subscripted array variables.) The following syntax rules will therefore replace those of the same name in Section 2.2:

*De-referencing:*

$$\frac{V: \mathbf{var}[\tau]}{V: \mathbf{exp}[\tau]}$$

*Assignment:*

$$\frac{V: \mathbf{var}[\tau] \quad E: \mathbf{exp}[\tau]}{V := E: \mathbf{comm}}$$

De-referencing is an example of a *coercion*: a phrase of one type ( $\mathbf{var}[\tau]$ ) can be used as a phrase of another type ( $\mathbf{exp}[\tau]$ ) without requiring an explicit conversion operator.

Coercions are convenient for programmers, but introduce several theoretical problems. One is that the language no longer has the property that, for any phrase-type assignment  $\pi$ , a phrase  $P$  has at most one type in  $\pi$ ; for example, if  $\pi(\iota) = \mathbf{var}[\tau]$ , then  $\pi \vdash \iota: \mathbf{var}[\tau]$  and  $\pi \vdash \iota: \mathbf{exp}[\tau]$ . But, we will show in Section 6.1 that, in any phrase-type assignment, every phrase that has a type has a ‘most-general’ type, which coerces into every other type that the phrase can have.

Another potential problem with coercions is that there can be ambiguity when they are combined with generic constructions. In fact, if  $B: \mathbf{exp}[\mathbf{bool}]$  and  $V_0, V_1: \mathbf{var}[\tau]$ , the conditional variable

**if  $B$  then  $V_0$  else  $V_1$**

is a phrase of type  $\mathbf{exp}[\tau]$  in *two* ways: either as a de-referenced conditional variable, or as a conditional expression whose two ‘arms’ are expressions obtained by de-referencing variables. The two derivations are as follows:

$$\frac{B: \mathbf{exp}[\mathbf{bool}] \quad V_0: \mathbf{var}[\tau] \quad V_1: \mathbf{var}[\tau]}{\text{if } B \text{ then } V_0 \text{ else } V_1: \mathbf{var}[\tau]} \\ \text{if } B \text{ then } V_0 \text{ else } V_1: \mathbf{exp}[\tau]$$

and

$$\frac{B: \mathbf{exp}[\mathbf{bool}] \quad \frac{V_0: \mathbf{var}[\tau]}{V_0: \mathbf{exp}[\tau]} \quad \frac{V_1: \mathbf{var}[\tau]}{V_1: \mathbf{exp}[\tau]}}{\text{if } B \text{ then } V_0 \text{ else } V_1: \mathbf{exp}[\tau]}$$

But it will turn out that there is no *semantic* ambiguity here, and so all is well. The question will be considered for the language as a whole in Section 6.1.

It is also convenient to introduce the following coercion to allow *any* phrase of type  $\mathbf{val}[\tau]$  to be used in contexts where phrases of type  $\mathbf{exp}[\tau]$  are expected, such as the right-hand sides of assignments:

*Constant expression:*

$$\frac{K: \mathbf{val}[\tau]}{K: \mathbf{exp}[\tau]}$$

Finally, we consider the conditional form

$$\mathbf{if } B \mathbf{ then } X_0 \mathbf{ else } X_1$$

with  $B: \mathbf{exp}[\mathbf{bool}]$  and  $X_0, X_1: \theta$ , which was originally defined for  $\theta$  ranging over  $\mathbf{exp}[\tau]$  and  $\mathbf{comm}$  only. Ideally we would want  $\theta$  to range now over *all* phrase types. For example, the following equivalence shows how to deal with procedural types (when the result type is one for which a conditional can be defined):

$$(\mathbf{if } B \mathbf{ then } X_1 \mathbf{ else } X_2)(A) \equiv_{\theta'} \mathbf{if } B \mathbf{ then } X_1(A) \mathbf{ else } X_2(A),$$

where  $X_1, X_2: \theta'' \rightarrow \theta'$  and  $A: \theta''$ . However, it does not seem to be feasible to have such a conditional form for phrases of type  $\theta$  when  $\theta$  is a 'purely-applicative' phrase type such as  $\mathbf{val}[\tau]$  or  $\theta' \rightarrow \mathbf{val}[\tau]$ , because the Boolean condition is state-dependent in general. Hence, in the following syntax rule

*Expression conditional:*

$$\frac{B: \mathbf{exp}[\mathbf{bool}] \quad X_0: \theta \quad X_1: \theta}{\mathbf{if } B \mathbf{ then } X_0 \mathbf{ else } X_1: \theta}$$

we must require that  $\theta$  *not* be a *value-like* phrase type, where the value-like phrase types are as follows:  $\mathbf{val}[\tau]$  for any data type  $\tau$ , and any functional phrase type of the form  $\theta'' \rightarrow \theta'$  when  $\theta'$  is value-like. The following syntax rule is applicable with *any* phrase type  $\theta$ :

*Value conditional:*

$$\frac{K: \mathbf{val}[\mathbf{bool}] \quad X_0: \theta \quad X_1: \theta}{\mathbf{if } K \mathbf{ then } X_0 \mathbf{ else } X_1: \theta}$$

but the Boolean condition  $K$  here must be state-independent because it is of type  $\mathbf{val}[\mathbf{bool}]$ .

We take *programs* to be elements of  $[\mathbf{comm}]_{\pi_0}$ ; i.e., phrases that are well-formed as commands in the initial phrase-type assignment for pre-defined identifiers. This completes the syntactic description of the initial fragment of our Algol-like language.

Our language allows programming in both the imperative style of the language of Section 2 and the applicative style of the language of Sections 3 and 4, and furthermore allows these styles to be used *together*. For example, the following defines a factorial-computing procedure recursively:

```

letrec fact(n: exp[nat], f: var[nat]): comm =
  if n = 0 then f := 1 else fact(n - 1, f) ; f := n × f
in ...

```

The following is semantically equivalent to this (except for the use of a non-local counter variable *k*), but uses iteration rather than recursion:

```

let fact(n: exp[nat], f: var[nat]) =
  k := 0 ; f := 1 ;
  while k ≠ n do
    k := k + 1 ; f := k × f
in ...

```

A facility for declaring *local* variables will be introduced in Section 6.2.

Here is an example involving a parameter of type **comm**:

```

let repeat(b: exp[bool], c: comm) =
  c ; while not b do c
in ...

```

Notice that, if this is to work as expected, the actual parameter corresponding to expression parameter *b* must be evaluated after *each* execution of the command parameter, *c*; in other words, parameters should be passed ‘by name’.

Finally, we present an example with a procedural parameter:

```

letrec iterate(a: exp[nat], b: exp[nat], p: exp[nat] → comm): comm =
  if a ≤ b then p(a) ; iterate(succ a, b, p)
in ...

```

The intended effect of *iterate*(*a*, *b*, *p*) is to apply *p* successively to *a*, *a* + 1, ..., *b*. For example,

$$\text{iterate}(a, b, \lambda i: \text{exp}[\text{nat}]. s := s + i \times i)$$

or, alternatively,

$$\begin{aligned} & \# \text{iterate}(a, b) \ i. \\ & s := s + i \times i \end{aligned}$$

have the overall effect of adding  $\sum_{i=a}^b i^2$  to a variable *s* of type **var**[nat].



## 5.2 Semantics

The semantic interpretation of the combined language can be obtained by combining the semantics of the simple imperative and applicative languages. The following domain definitions are essentially taken from the preceding sections:

$$\begin{aligned}
 \llbracket \mathbf{val}[\tau] \rrbracket &= \llbracket \tau \rrbracket_{\perp} \\
 \llbracket \mathbf{exp}[\tau] \rrbracket &= S \rightarrow \llbracket \mathbf{val}[\tau] \rrbracket \\
 \llbracket \mathbf{comm} \rrbracket &= S \multimap S \\
 \llbracket \theta \rightarrow \theta' \rrbracket &= \llbracket \theta \rrbracket \rightarrow \llbracket \theta' \rrbracket \\
 \llbracket (\theta) \rrbracket &= \llbracket \theta \rrbracket \\
 \llbracket \pi \rrbracket &= \prod_{\iota \in \text{dom } \pi} \llbracket \pi(\iota) \rrbracket,
 \end{aligned}$$

where, as before,  $S$  is the set of states (to be defined below), and, for every data type  $\tau$ ,  $\llbracket \tau \rrbracket$  is the set of  $\tau$ -values. The only notable change is that evaluation of a phrase of type  $\mathbf{exp}[\tau]$  at some state can now yield  $\perp$  (signifying non-termination) as well as an element of  $\llbracket \tau \rrbracket$ ; in Section 2 we could use  $\llbracket \mathbf{exp}[\tau] \rrbracket = S \rightarrow \llbracket \tau \rrbracket$  because expression evaluations there always terminated.

Notice that all functional and procedural types  $\theta \rightarrow \theta'$  are treated uniformly. As an example of the interpretation of a procedural type, consider

$$\llbracket \mathbf{exp}[\mathbf{nat}] \rightarrow \mathbf{comm} \rrbracket = (S \rightarrow N_{\perp}) \rightarrow (S \multimap S);$$

i.e., the domain of continuous functions from meanings of expressions for natural numbers to command meanings.

We must also define  $\llbracket \mathbf{var}[\tau] \rrbracket$ . To allow for conditional variables and recursive definitions of variables and variable-returning procedures, and analogously to

$$\llbracket \mathbf{exp}[\tau] \rrbracket = S \rightarrow \llbracket \tau \rrbracket_{\perp},$$

we define

$$\llbracket \mathbf{var}[\tau] \rrbracket = S \rightarrow [\tau]_{\perp},$$

where

$$[\tau] = \{ \iota \in \text{dom } \pi_0 \mid \pi_0(\iota) = \mathbf{var}[\tau] \}$$

is the set of pre-defined  $\tau$ -valued variable identifiers; that is, a phrase of type  $\mathbf{var}[\tau]$  denotes a function that, given any state, yields  $\perp$  (signifying non-termination) or a pre-defined variable-identifier for values of type  $\tau$ . As before, a state is any function  $s$  whose domain is the set of all identifiers  $\iota$  such that  $\pi_0(\iota) = \mathbf{var}[\tau]$  for some data type  $\tau$ , and  $s(\iota) \in \llbracket \tau \rrbracket$  for all  $\iota \in [\tau]$ ;  $S$  is the set of all such states.

We can now define the semantic valuation functions

$$\llbracket \cdot \rrbracket_{\pi\theta} : [\theta]_{\pi} \rightarrow ([\pi] \rightarrow [\theta]),$$

where, as before,  $[\theta]_{\pi}$  is the set of all phrases  $X$  such that  $\pi \vdash X : \theta$ . For the applicative aspects of the language, the semantic equations in Sections 3 and 4 can be used without any changes whatsoever, even when imperative phrases such as variables or commands are involved. The most important of these equations are as follows:

$$\begin{aligned} \llbracket \iota \rrbracket u &= u(\iota) \\ \llbracket P Q \rrbracket u &= \llbracket P \rrbracket u (\llbracket Q \rrbracket u) \\ \llbracket \lambda \iota : \theta. P \rrbracket u a &= \llbracket P \rrbracket (u \mid \iota \mapsto a), \text{ for all } a \in [\theta] \\ \llbracket \text{undef}[\theta] \rrbracket u &= \perp_{\theta} \\ \llbracket \text{rec}[\theta] \rrbracket u f &= \bigsqcup_{i \in \omega} f^i(\perp_{\theta}), \text{ for all } f \in [\theta] \rightarrow [\theta] \\ \llbracket \text{if } K \text{ then } X_0 \text{ else } X_1 \rrbracket_{\theta}(u) &= \begin{cases} \llbracket X_0 \rrbracket_{\theta}(u), & \text{if } \llbracket K \rrbracket u = \text{true} \\ \llbracket X_1 \rrbracket_{\theta}(u), & \text{if } \llbracket K \rrbracket u = \text{false} \\ \perp_{\theta}, & \text{if } \llbracket K \rrbracket u = \perp \end{cases} \end{aligned}$$

where  $K : \mathbf{val}[\mathbf{bool}]$ . Notice that if the actual parameter ( $Q$ ) in a procedure application  $P Q$  is an expression, variable, or command, it is not evaluated (or executed) by the call (the state argument is not supplied); i.e., ‘lazy evaluation’ or call-by-name parameter passing is specified.

For the imperative aspects, the semantic equations of Sections 2.1 and 2.3 need to be changed only by adding environment arguments to valuations throughout to account for identifier binding, and allowing for  $\perp$  as a possible result of sub-expression evaluation, as in the following:

$$\begin{aligned} \llbracket \text{not } B \rrbracket u s &= \begin{cases} \text{false}, & \text{if } \llbracket B \rrbracket u s = \text{true} \\ \text{true}, & \text{if } \llbracket B \rrbracket u s = \text{false} \\ \perp, & \text{if } \llbracket B \rrbracket u s = \perp \end{cases} \\ \llbracket \text{skip} \rrbracket u &= \text{id}_S \\ \llbracket C_0 ; C_1 \rrbracket u &= \llbracket C_0 \rrbracket u ; \llbracket C_1 \rrbracket u \\ \llbracket \text{diverge} \rrbracket u &= \perp_{\text{comm}} \\ \llbracket \text{while } B \text{ do } C \rrbracket u &= \bigsqcup_{i \in \omega} f^i(\perp_{\text{comm}}), \\ \text{where } f(c)(s) &= \begin{cases} (\llbracket C \rrbracket u ; c)(s), & \text{if } \llbracket B \rrbracket u s = \text{true} \\ s, & \text{if } \llbracket B \rrbracket u s = \text{false} \\ \text{undefined}, & \text{if } \llbracket B \rrbracket u s = \perp \end{cases} \end{aligned}$$

To interpret the expression-conditional forms we must resort to an induction on types as follows: for  $B : \mathbf{exp}[\mathbf{bool}]$  and  $X_0, X_1 : \theta$  ( $\theta$  not value-like),

$$\llbracket \text{if } B \text{ then } X_0 \text{ else } X_1 \rrbracket_{\theta}(u) = \text{cond}_{\theta}(\llbracket B \rrbracket u, \llbracket X_0 \rrbracket_{\theta}(u), \llbracket X_1 \rrbracket_{\theta}(u))$$

where the auxiliary functions  $cond_\theta: \llbracket \mathbf{exp}[\mathbf{bool}] \rrbracket \times \llbracket \theta \rrbracket \times \llbracket \theta \rrbracket \rightarrow \llbracket \theta \rrbracket$  are defined by induction on the structure of  $\theta$  as follows:

$$\begin{aligned} cond_{\mathbf{exp}[\tau]}(b, e_0, e_1)(s) &= \begin{cases} e_0(s), & \text{if } b(s) = \text{true} \\ e_1(s), & \text{if } b(s) = \text{false} \\ \perp, & \text{if } b(s) = \perp \end{cases} \\ cond_{\mathbf{var}[\tau]}(b, v_0, v_1)(s) &= \begin{cases} v_0(s), & \text{if } b(s) = \text{true} \\ v_1(s), & \text{if } b(s) = \text{false} \\ \perp, & \text{if } b(s) = \perp \end{cases} \\ cond_{\mathbf{comm}}(b, c_0, c_1)(s) &= \begin{cases} c_0(s), & \text{if } b(s) = \text{true} \\ c_1(s), & \text{if } b(s) = \text{false} \\ \text{undefined}, & \text{if } b(s) = \perp \end{cases} \\ cond_{\theta \rightarrow \theta'}(b, p_0, p_1)(a) &= cond_{\theta'}(b, p_0(a), p_1(a)), \text{ for all } a \in \llbracket \theta \rrbracket, \end{aligned}$$

where  $cond_{\theta'}$  must exist because  $\theta'$  is not value-like.

The semantic equations for constant expressions, assignment commands, and de-referencing are as follows:

$$\begin{aligned} \llbracket K \rrbracket_{\mathbf{exp}[\tau]}(u)(s) &= \llbracket K \rrbracket_{\mathbf{val}[\tau]}(u) \\ \llbracket V := E \rrbracket_{us} &= \begin{cases} (s \mid \iota \mapsto n), & \text{if } \llbracket V \rrbracket_{\mathbf{var}[\tau]}(u)(s) = \iota \in [\tau] \text{ and } \llbracket E \rrbracket_{us} = n \in [\tau] \\ \text{undefined}, & \text{if } \llbracket V \rrbracket_{\mathbf{var}[\tau]}(u)(s) = \perp \text{ or } \llbracket E \rrbracket_{us} = \perp \end{cases} \\ \llbracket V \rrbracket_{\mathbf{exp}[\tau]}(u)(s) &= \begin{cases} s(\iota), & \text{if } \llbracket V \rrbracket_{\mathbf{var}[\tau]}(u)(s) = \iota \in [\tau] \\ \perp, & \text{if } \llbracket V \rrbracket_{\mathbf{var}[\tau]}(u)(s) = \perp \end{cases} \end{aligned}$$

for any  $s \in S$ .

**Exercise 5.2.1.** Verify that both derivations of

$$\mathbf{if } B \mathbf{ then } V_0 \mathbf{ else } V_1: \mathbf{exp}[\tau]$$

from  $B: \mathbf{exp}[\mathbf{bool}]$ ,  $V_0: \mathbf{var}[\tau]$ , and  $V_1: \mathbf{var}[\tau]$  yield the same interpretation.

We define the meanings of variable-identifiers in the ‘initial’ environment  $u_0$  as follows: for all  $\iota \in [\tau]$  and  $s \in S$ ,  $u_0(\iota)(s) = \iota$ . This gives us essentially the same semantics as before for variable-identifiers.

This completes the semantic interpretation of the Algol-like language obtained by combining the facilities of the simple imperative and applicative languages of the preceding sections. It is noteworthy that the language retains virtually all of the logical properties of the ‘pure’ languages, as shown by the following facts.

- All of the command equivalences of Sections 2.1, 2.2 and 2.3 continue to be valid.
- The Coherence and Substitution lemmas and the other propositions of Section 3.4 continue to hold.

- All of the axioms of Sections 3.5 and 4.4 (including Substitutivity and the Beta and Eta laws) are valid, even for imperative phrases such as variables or commands.
- We will see in Section 5.4 that, with only one exception, all of the Hoare-logic axioms of Section 2.6.2 remain valid with only minor modifications.

We also have some additional equational axioms relating conditionals and variables or procedures. When  $B \in [\mathbf{exp}[\mathbf{bool}]]_\pi$ ,  $V_0, V_1 \in [\mathbf{var}[\tau]]_\pi$ , and  $E \in [\mathbf{exp}[\tau]]_\pi$ ,

$$\pi \vdash (\mathbf{if } B \mathbf{ then } V_0 \mathbf{ else } V_1) := E \equiv_{\mathbf{comm}} \mathbf{if } B \mathbf{ then } V_0 := E \mathbf{ else } V_1 := E$$

When  $B \in [\mathbf{exp}[\mathbf{bool}]]_\pi$ ,  $X_1, X_2 \in [\theta \rightarrow \theta']_\pi$  and  $A \in [\theta]_\pi$ ,

$$\pi \vdash (\mathbf{if } B \mathbf{ then } X_1 \mathbf{ else } X_2)(A) \equiv_{\theta'} \mathbf{if } B \mathbf{ then } X_1(A) \mathbf{ else } X_2(A),$$

and, when  $X_1, X_2 \in [\theta']_{(\pi|\iota \mapsto \theta)}$ ,  $B \in [\mathbf{exp}[\mathbf{bool}]]_\pi$ , and  $\iota \notin \text{dom } \pi$ ,

$$\pi \vdash \lambda\iota: \theta. \mathbf{if } B \mathbf{ then } X_1 \mathbf{ else } X_2 \equiv_{\theta \rightarrow \theta'} \mathbf{if } B \mathbf{ then } \lambda\iota: \theta. X_1 \mathbf{ else } \lambda\iota: \theta. X_2,$$

and similarly if  $B: \mathbf{val}[\mathbf{bool}]$ .

**Exercise 5.2.2.** Define an operational semantics for the language described in this section and verify its correctness.

### 5.3 Call by value

Two of the most important design principles underlying the language described so far are as follows.

- As in the applicative languages, parameter passing is ‘by name’, so that an actual parameter is evaluated (or executed) whenever its value or effect is actually needed by the calling procedure (if ever), rather than at the call.
- As in the simple imperative language, commands and expressions are distinguished in that commands can change the computational state but do not have values, and expressions can have values but cannot change the state.

Although virtually all practical programming languages combine applicative and imperative aspects, few of them adhere to these design principles.

To justify our apparently eccentric choice of example language and for comparison, we briefly discuss in this section a language that illustrates the

combination of applicative and imperative features according to different language-design principles, as follows.

- ‘Eager’ evaluation (call by value), in which an actual parameter is always evaluated *exactly once* before evaluation of the calling procedure, is usually more efficient than ‘lazy’ evaluation (call by name).
- If expression ‘evaluation’ can *change* (as well as access) the state, the type structure can be simplified by treating commands and expressions uniformly.

The syntactic-type structure for the language we consider in this section is as follows:

$$\begin{array}{ll} \tau ::= \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{nat} & \text{data types} \\ \theta ::= \mathbf{exp}[\tau] \mid \mathbf{var}[\tau] \mid \theta \rightarrow \theta' \mid (\theta) & \text{phrase types} \end{array}$$

Expressions of type **unit** always have a ‘dummy’ value *nil*; they are normally ‘evaluated’ solely for their effect on the state. For example, assignments here are expressions of type **unit**:

*Assignment:*

$$\frac{V: \mathbf{var}[\tau] \quad E: \mathbf{exp}[\tau]}{V := E: \mathbf{exp}[\mathbf{unit}]}$$

The semi-colon operator is generalized to allow sequential evaluation of arbitrary expressions:

*Sequencing:*

$$\frac{E_0: \mathbf{exp}[\tau_0] \quad E_1: \mathbf{exp}[\tau_1]}{E_0 ; E_1: \mathbf{exp}[\tau_1]}$$

The value is that of  $E_1$  after evaluation of  $E_0$  for its effect on the computational state; the value of  $E_0$  is simply discarded.

We now describe the semantics of the language. As before, we use environments for the applicative aspects and states for the imperative aspects. We interpret the data types as follows:

$$\begin{aligned} \llbracket \mathbf{unit} \rrbracket &= \{\mathit{nil}\} \\ \llbracket \mathbf{bool} \rrbracket &= \{\mathit{true}, \mathit{false}\} \\ \llbracket \mathbf{nat} \rrbracket &= \{0, 1, 2, \dots\} \end{aligned}$$

For phrase types, the call-by-value regime forces us to distinguish between the *denotable* and the *expressible* entities of each phrase type. The former are the values denoted by identifiers of that phrase type (after evaluation); the latter are the meanings expressed by phrases of that type (before evaluation). We continue to use  $\llbracket \theta \rrbracket$  for the domain of *denotable* values of phrase type  $\theta$ , and define the operation



$$\mathcal{E}(D) = S \rightarrow D \times S$$

on domains  $D$ , so that  $\mathcal{E}[\theta]$  is the domain of *expressible* meanings of phrase type  $\theta$ ; as usual,  $S$  is an appropriate set of states. This definition allows evaluation at some state either to fail to terminate or to terminate with a value and a possibly-updated state.

The domain of denotable values for any phrase type  $\theta$  and the domain of environments appropriate to any phrase-type assignment  $\pi$  are defined as follows:

$$\begin{aligned} \llbracket \mathbf{exp}[\tau] \rrbracket &= \llbracket \tau \rrbracket \\ \llbracket \mathbf{var}[\tau] \rrbracket &= \llbracket \tau \rrbracket \\ \llbracket \theta \rightarrow \theta' \rrbracket &= \llbracket \theta \rrbracket \rightarrow \mathcal{E}[\theta'] \\ \llbracket (\theta) \rrbracket &= \llbracket \theta \rrbracket \\ \llbracket \pi \rrbracket &= \prod_{\iota \in \text{dom } \pi} \llbracket \pi(\iota) \rrbracket, \end{aligned}$$

where  $\llbracket \tau \rrbracket$  is, as before, the set of pre-defined  $\tau$ -valued variable identifiers. Notice that an identifier of type  $\theta \rightarrow \theta'$  denotes a function mapping each *denotable* value of type  $\theta$  to an *expressible* meaning of type  $\theta'$ .

The valuation functions map phrases into continuous functions from appropriate environments to *expressible* meanings of appropriate type, as follows; for every phrase-type assignment  $\pi$  and phrase type  $\theta$ ,

$$\llbracket \cdot \rrbracket_{\pi\theta} : [\theta]_{\pi} \rightarrow (\llbracket \pi \rrbracket \rightarrow \mathcal{E}[\theta]).$$

Here are some typical semantic equations; as before,  $u$  and  $s$  range over environments and states, respectively.

$$\begin{aligned} \llbracket 0 \rrbracket us &= (0, s) \\ \llbracket \mathbf{skip} \rrbracket us &= (\text{nil}, s) \\ \llbracket \mathbf{diverge} \rrbracket us &= \text{undefined} \\ \llbracket \iota \rrbracket us &= (u(\iota), s) \\ \llbracket \lambda \iota : \theta. P \rrbracket us &= (f, s), \text{ where } f(a) = \llbracket P \rrbracket (u \mid \iota \mapsto a) \text{ for all } a \in [\theta] \\ \llbracket PQ \rrbracket us &= \begin{cases} f(a)(s''), & \text{if } \llbracket P \rrbracket us = (f, s') \text{ and } \llbracket Q \rrbracket us' = (a, s'') \\ \text{undefined}, & \text{otherwise} \end{cases} \\ \llbracket V := E \rrbracket us &= \begin{cases} (\text{nil}, (s'' \mid \iota \mapsto v)), & \text{if } \llbracket V \rrbracket us = (\iota, s') \text{ and } \llbracket E \rrbracket us' = (v, s'') \\ \text{undefined}, & \text{otherwise} \end{cases} \\ \llbracket E_0 ; E_1 \rrbracket us &= \begin{cases} \llbracket E_1 \rrbracket us', & \text{if } \llbracket E_0 \rrbracket us = (v, s') \\ \text{undefined}, & \text{otherwise} \end{cases} \\ \llbracket \mathbf{if } B \mathbf{ then } X_0 \mathbf{ else } X_1 \rrbracket us &= \begin{cases} \llbracket X_0 \rrbracket us', & \text{if } \llbracket B \rrbracket us = (\text{true}, s') \\ \llbracket X_1 \rrbracket us', & \text{if } \llbracket B \rrbracket us = (\text{false}, s') \\ \text{undefined}, & \text{otherwise} \end{cases} \end{aligned}$$

Evaluation of identifiers, abstractions, and constants such as **skip** and 0 yield values and the *same* state; other expressions, such as the assignment, can yield *new* states. In general, *any* sub-expression evaluation might, as a *side effect*, change the computational state or fail to terminate, as well as yield a value, and these possibilities must be ‘propagated’. Notice that the actual parameter in the procedure-application form is evaluated before the procedure is applied to the resulting argument value. This captures the call-by-value regime; for example,  $\llbracket P(\text{diverge}) \rrbracket us$  is always undefined.

This completes the semantic description of the call-by-value language. Although it is only slightly more complicated than that for the Algol-like language of the preceding sections, its *logical* properties are very different. The key difference is that, because of the distinction between denotable values and expressible meanings, the Substitution lemma fails. This means that none of the logical rules or axioms (such as  $\forall$ -elimination) whose validity depend on this lemma can be used without restrictions. Another important difference is that expression evaluation can have side effects on the computational state, and this means that conventional ‘algebraic’ reasoning cannot be used with expressions. Furthermore, it does not seem possible to reason about assignments using substitution of an expression for a variable in an assertion, because we cannot deal with side effects in an assertion.

These difficulties are the subjects of current research. It remains to be seen whether the *operational* advantage of call-by-value can be obtained without significant *logical* cost.

In summary, many programming languages (LISP, ALGOL 68, SCHEME, ML, ...) use variants of lambda notation for functions or procedures; but only ALGOL 60, which does not even have lambda expressions, and its very close relations (such as the language described in the preceding sections) combine applicative and imperative features in a way that preserves the *logical principles*, and not merely the *notation*, of the lambda calculus.

**Exercise 5.3.1.** Discuss the syntax and semantics of

1. recursion, and
2. multi-parameter functions

in the language of this section.

**Exercise 5.3.2.** Define an operational semantics for the call-by-value language described in this section and verify its correctness.

## 5.4 Programming logic

We now consider how Hoare’s logic for the simple imperative language (Section 2.6) can be adapted to the Algol-like language of this section, which has both commands and procedures.

### 5.4.1 Syntax and semantics

We first consider how assertions (phrase type **assert**) can be treated. In Section 2.6, assertions and Boolean expressions were regarded as identical. But this would be difficult now, because evaluation of a Boolean expression may fail to terminate. Instead, we continue to use

$$\llbracket \text{assert} \rrbracket = S \rightarrow \llbracket \text{bool} \rrbracket,$$

so that an assertion yields a truth value in every state, whereas

$$\llbracket \text{exp}[\text{bool}] \rrbracket = S \rightarrow \llbracket \text{bool} \rrbracket_{\perp}.$$

For example, instead of the doubly-strict interpretation used for the *Boolean expression*  $N < N'$ , we define the *assertion*  $N < N'$  as follows:

$$\begin{aligned} & \llbracket N < N' \rrbracket_{\text{assert}}(u)(s) \\ &= \begin{cases} \text{true}, & \text{if } \llbracket N \rrbracket us \neq \perp, \llbracket N' \rrbracket us \neq \perp, \llbracket N \rrbracket us < \llbracket N' \rrbracket us; \\ \text{false}, & \text{otherwise,} \end{cases} \end{aligned}$$

and similarly for the other primitive assertions. This approach ‘localizes’ the problems created by non-termination as much as possible; in particular, the assertional operators **not**, **and**,  $\supset$ , and so on can be treated exactly as before.

We will treat equality assertions in the same way, so that  $\llbracket E = E' \rrbracket_{\text{assert}}(u)(s) = \text{true}$  just if both operands have the same value which is *not*  $\perp$ . For some applications, an equality assertion  $E \equiv E'$  over the *whole* domain, including  $\perp$ , is needed. This can be added to the syntax as follows:

*Reflexive equality:*

$$\frac{E: \text{exp}[\tau] \quad E': \text{exp}[\tau]}{E \equiv E': \text{assert}}$$

and interpreted as follows:

$$\llbracket E \equiv E' \rrbracket_{\text{assert}}(u)(s) = (\llbracket E \rrbracket us = \llbracket E' \rrbracket us)$$

for all  $s \in S$ . Note that the assertion  $E = E$  is *true* exactly when the evaluation of  $E$  terminates, but  $E \equiv E$  is *true* in *every* state.

By introducing suitable constants **forall** $[\tau]$  and **exists** $[\tau]$  of phrase type  $(\text{val}[\tau] \rightarrow \text{assert}) \rightarrow \text{assert}$  for every data type  $\tau$ , we can obtain universal and existential quantification in the assertion language as follows:

$$\begin{aligned} \# \text{forall}[\tau] \iota. P \\ \# \text{exists}[\tau] \iota. P \end{aligned}$$

An appropriate interpretation of these quantifier constants is as follows; for all environments  $u$ ,  $f \in \llbracket \text{val}[\tau] \rrbracket \rightarrow \llbracket \text{assert} \rrbracket$ , and states  $s$ ,

$$\begin{aligned}\llbracket \text{forall}[\tau] \rrbracket ufs &= (\text{for all } v \in \llbracket \text{val}[\tau] \rrbracket, f(v)(s) = \text{true}) \\ \llbracket \text{exists}[\tau] \rrbracket ufs &= (\text{for some } v \in \llbracket \text{val}[\tau] \rrbracket, f(v)(s) = \text{true})\end{aligned}$$

This gives us, for example,

$$\begin{aligned}\llbracket \# \text{forall}[\tau] \iota. P \rrbracket us &= \llbracket \text{forall}[\tau] (\lambda \iota. \text{val}[\tau]. P) \rrbracket us \\ &= \llbracket \text{forall}[\tau] \rrbracket u(\llbracket \lambda \iota. \text{val}[\tau]. P \rrbracket u)s \\ &= (\text{for all } v \in \llbracket \text{val}[\tau] \rrbracket, \llbracket P \rrbracket (u \mid \iota \mapsto v)s = \text{true})\end{aligned}$$

and similarly for **exists** $[\tau]$ .

Specification formulas can be treated as in Sections 3.5 and 4.4, but using Hoare triples as the atomic formulas; the semantic equation for Hoare triples in this context is as follows:

$$\begin{aligned}\llbracket \{P\}C\{Q\} \rrbracket u \\ = \text{for all } s_0, s_1 \in S, \text{ if } \llbracket P \rrbracket us_0 \text{ and } \llbracket C \rrbracket us_0 = s_1 \text{ then } \llbracket Q \rrbracket us_1\end{aligned}$$

An environment argument has been added to each valuation to handle free identifiers.

### 5.4.2 Axioms

We now discuss axioms for reasoning about Hoare-triple specifications for our combined language. A minor problem with two of the axioms of Section 2.6.2 is that they are no longer syntactically well-formed because we now distinguish between assertions and Boolean expressions; hence, we must revise the axioms for the conditional and iterative forms of command as follows:

$$\begin{aligned}&\{P \text{ and } (B = \text{true})\}C_0\{Q\} \ \& \\ &\{P \text{ and } (B = \text{false})\}C_1\{Q\} \Rightarrow \\ &\{P\}\text{if } B \text{ then } C_0 \text{ else } C_1\{Q\}\end{aligned}$$

$$\begin{aligned}&\{P \text{ and } (B = \text{true})\}C\{P\} \Rightarrow \\ &\{P\}\text{while } B \text{ do } C\{P \text{ and } (B = \text{false})\}\end{aligned}$$

With one exception, all of the other axioms presented in Section 2.6.2 remain valid without any changes. Unfortunately, the exception is the most fundamental axiom in Hoare-logic reasoning, the assignment axiom:

$$\{[P](\iota \mapsto E)\}_\iota := E\{P\}.$$

There are essentially two difficulties.

1. It is no longer true that assigning to one variable identifier cannot affect any other variable identifier. For example, suppose  $b$ : **var**[**bool**] and consider

**let**  $c$  **be**  $b$  **in**  
 $b := \mathbf{not} \ b; \dots$

then, the value of  $c$  is affected by the assignment to  $b$  because  $c$  is an *alias* for the same variable in the scope of the definition. The assignment axiom of Section 2.6.2 would allow us to derive

$$\{[c = \mathbf{not} \ b](b \mapsto \mathbf{not} \ b)\}b := \mathbf{not} \ b\{c = \mathbf{not} \ b\}$$

which is

$$\{c = \mathbf{not} \ \mathbf{not} \ b\}b := \mathbf{not} \ b\{c = \mathbf{not} \ b\}$$

and this is clearly invalid if  $b$  and  $c$  denote the *same* variable.

A more subtle kind of aliasing can arise from the use of non-local variables or procedures in procedure bodies. For example, suppose  $m$ : **var**[**nat**] and consider

**let**  $addm(x: \mathbf{exp}[\mathbf{nat}]) = x + m$   
**in**  $\dots$

Any assignment to  $m$  interferes with assertions that involve  $addm$ ; for example, the following instance of Hoare's assignment axiom

$$\{a = addm(b)\}m := \mathbf{succ} \ m\{a = addm(b)\}$$

is clearly invalid.

2. It is no longer true in general that, immediately after assigning to a variable, the value of that variable is the value just assigned! For example, suppose that  $b0, b1$ : **var**[**bool**] and consider the assignment  $B := \mathbf{true}$  when  $B$  is the conditional variable

**if**  $b1$  **then**  $b0$  **else**  $b1$

If  $b1$  is *false* initially, the assignment is equivalent to  $b1 := \mathbf{true}$ , and so the value of the conditional variable after the assignment is the value of  $b0$ , which might happen to be *false*.

A variable phrase  $V \in [\mathbf{var}[\tau]]_\pi$  is termed *good* just if, for all  $E \in [\mathbf{exp}[\tau]]_\pi$ ,  $u \in \llbracket \pi \rrbracket$ , and  $s \in S$ , if  $\llbracket V := E \rrbracket us = s'$ , then

$$\llbracket V \rrbracket_{\pi \mathbf{exp}[\tau]}(u)(s') = \llbracket E \rrbracket us;$$



that is, immediately after assigning to a good variable, the value of the variable is the value just assigned. The difficulty with conditional variable phrases is that they may not be good variables, even when their two sub-variables are good variables. The same kind of difficulty arises whenever variable phrases can have expressions as components; in a language with subscripted array variables, for example,  $a[a[i]]$  is not a good variable because, when  $a[i] = i$ , an assignment to  $a[a[i]]$  may change the value of the outer subscript.

There are basically two approaches to these problems. One is to design a language in which aliasing and bad variables can easily be precluded or detected *syntactically*; the other is to introduce new atomic specification formulas that allow relevant assumptions about variables and assertions to be stated and reasoned about in the programming logic. The first approach is beyond the scope of this chapter, but we will consider the second approach in Section 7.6 and give there a generally valid axiom for assignments in our language.

**Exercise 5.4.1.** The following axiom for assignment commands was proposed in [Floyd, 1967]:

$$\{P\} \iota := E \{ \# \text{ exists } [\tau]_{\iota_0}. \iota = [E](\iota \mapsto \iota_0) \text{ and } [P](\iota \mapsto \iota_0) \}$$

where  $\iota \in [\tau]$  and  $\iota_0$  is any identifier both distinct from  $\iota$  and not free in  $E$  or  $P$ .

1. Verify the validity of this axiom for the simple imperative language of Section 2.
2. Is this axiom valid for the Algol-like language discussed in this section?

## 5.5 Bibliographic notes

The key reference for this section and the following one is [Reynolds, 1981b]; see also Reynolds [1978; 1981a; 1982] and [Tennent, 1989]. The semantics and logic of call-by-value languages are discussed in [Landin, 1966; Plotkin, 1975; Rosolini, 1986; Felleisen, 1987; Felleisen and Friedman, 1989; Moggi, 1988; Moggi, 1989; Riecke, 1990; Pitts, 1991; Mason and Talcott, 1991; Mason and Talcott, 1992]. The treatment of Boolean expressions and assertions is from [Tennent, 1987a]. A syntactic approach to the problems of aliasing and interference in Algol-like languages is discussed in [Reynolds, 1978; Tennent, 1983; Reynolds, 1989; O'Hearn, 1991; O'Hearn, 1993].

## 6 An Algol-like language II

In this section, we continue the design of the Algol-like language whose 'core' was discussed in Section 5. We begin with a detailed consideration of coercions, addressing the questions raised about syntactic and semantic ambiguity. We then go on to consider local-variable declarations, data

structures, ‘write-only variables’, jumps, and block expressions. Many of these discussions will illustrate the usefulness of rigorous semantic analysis in language design.

## 6.1 Coercions

Many language designers in recent years have tried to avoid implicit conversions as much as possible because of the many difficulties and surprises encountered by programmers using languages such as PL/I and ALGOL 68, which had very complex systems of coercions. But coercions are too convenient to abandon completely: every imperative high-level programming language has at least de-referencing and a coercion from integer to real-valued expressions. The aim of this section is to describe a relatively simple approach to coercions for our Algol-like language and verify that it has appropriate theoretical properties.

We can avoid one kind of syntactic complexity by deciding that the applicability of any coercion from one type to another should not be dependent on syntactic context. Then coercibility can be specified by means of a binary relation on phrase-type expressions, which we denote by  $\vdash$ . The relationship  $\theta \vdash \theta'$  may be regarded as an abbreviation for the syntax rule

$$\frac{X: \theta}{X: \theta'}$$

i.e., for any type assignment  $\pi$  and phrase  $X$ ,  $\pi \vdash X: \theta$  implies  $\pi \vdash X: \theta'$ , so that, in any context where phrases of type  $\theta'$  are expected, any phrase of type  $\theta$  can be used as well. Sometimes  $\theta$  is termed a ‘sub-type’ of  $\theta'$ ; however, a ‘sub-type’ need not be interpreted as a sub-set. For example, consider the following coercions, which we have already discussed:

$$\text{var}[\tau] \vdash \text{exp}[\tau]$$

$$\text{val}[\tau] \vdash \text{exp}[\tau]$$

for every data type  $\tau$ . The domains of variable and value meanings are not subsets of the domain of expression meanings.

It is natural to require reflexivity ( $\theta \vdash \theta$ ) and transitivity ( $\theta_1 \vdash \theta_2$  and  $\theta_2 \vdash \theta_3$  imply  $\theta_1 \vdash \theta_3$ ), so that  $\vdash$  is in general a pre-order. We write  $\theta \simeq \theta'$  if  $\theta \vdash \theta'$  and  $\theta' \vdash \theta$ ; it is easily proved that  $\simeq$  is an equivalence relation. Requiring that  $\vdash$  be anti-symmetric ( $\theta \simeq \theta'$  only if  $\theta = \theta'$ ) would ensure that  $\vdash$  is always a partial order, but this is undesirable if it forces an arbitrary choice of canonical representative for a  $\simeq$ -equivalence class.

Semantically, we need to define a *conversion* function

$$[[\theta \vdash \theta']]: [[\theta]] \rightarrow [[\theta']]$$

whenever  $\theta \vdash \theta'$ , so that, for any phrase  $X$  such that  $\pi \vdash X : \theta$ ,

$$\llbracket X \rrbracket_{\pi\theta'} = \llbracket X \rrbracket_{\pi\theta} ; \llbracket \theta \vdash \theta' \rrbracket$$

whenever  $\theta \vdash \theta'$ . For computational phrase types, we will require that these conversions be strict ( $\perp$ -preserving) and continuous. For example, the conversion for de-referencing may be defined as follows:

$$\llbracket \mathbf{var}[\tau] \vdash \mathbf{exp}[\tau] \rrbracket vs = \begin{cases} s(v(s)), & \text{if } v(s) \in [\tau] \\ \perp, & \text{if } v(s) = \perp. \end{cases}$$

To preclude ambiguity, we require that  $\llbracket \theta \vdash \theta \rrbracket$  be the identity function on  $\llbracket \theta \rrbracket$ , and that, when  $\theta_1 \vdash \theta_2$  and  $\theta_2 \vdash \theta_3$ ,  $\llbracket \theta_1 \vdash \theta_3 \rrbracket$  be equal to the composite conversion  $\llbracket \theta_1 \vdash \theta_2 \rrbracket ; \llbracket \theta_2 \vdash \theta_3 \rrbracket$ .

We will also want to define a non-trivial pre-order (also denoted  $\vdash$ ) on *data-type* names. For example, we can introduce new data-type symbols **int** and **real** with  $\llbracket \mathbf{int} \rrbracket$  and  $\llbracket \mathbf{real} \rrbracket$  as the sets of integer and real numbers, respectively, and then specify that  $\mathbf{nat} \vdash \mathbf{int} \vdash \mathbf{real}$ , with the corresponding conversions,  $\llbracket \mathbf{nat} \vdash \mathbf{int} \rrbracket$  and  $\llbracket \mathbf{int} \vdash \mathbf{real} \rrbracket$ , being the obvious injections. A Hasse diagram of the resulting data-type pre-order is then



We will then want every such data-type coercion  $\tau \vdash \tau'$  to induce the following coercions on *phrase* types:

$$\mathbf{val}[\tau] \vdash \mathbf{val}[\tau']$$

$$\mathbf{exp}[\tau] \vdash \mathbf{exp}[\tau']$$

For example, an integer-valued expression should be usable wherever an expression producing reals is, because any integer value it produces can be converted to a real number. In general:

$$\begin{aligned} \llbracket \mathbf{val}[\tau] \vdash \mathbf{val}[\tau'] \rrbracket &= \llbracket \tau \vdash \tau' \rrbracket_{\perp} \\ \llbracket \mathbf{exp}[\tau] \vdash \mathbf{exp}[\tau'] \rrbracket e &= e ; \llbracket \tau \vdash \tau' \rrbracket_{\perp}, \end{aligned}$$

for all  $e \in \llbracket \mathbf{exp}[\tau] \rrbracket$ , where  $\llbracket \tau \vdash \tau' \rrbracket_{\perp}$  is the strict extension of

$$\llbracket \tau \vdash \tau' \rrbracket : \llbracket \tau \rrbracket \rightarrow \llbracket \tau' \rrbracket$$

to  $\llbracket \tau \rrbracket_{\perp} \rightarrow \llbracket \tau' \rrbracket_{\perp}$ .

We can also introduce coercions on *functional* and *procedural* types; these are determined by the coercibility of their argument and result types as follows:

$$\theta_0 \rightarrow \theta_1 \vdash \theta'_0 \rightarrow \theta'_1 \text{ whenever } \theta'_0 \vdash \theta_0 \text{ and } \theta_1 \vdash \theta'_1.$$

Notice that the  $\rightarrow$  operator is monotone in its second operand but *anti-monotone* in its first operand. For example, a procedure expecting a **real**-valued phrase should be usable wherever it is possible to use a procedure with the same result type but expecting an **int**-valued phrase as an actual parameter, because an integer argument can be converted to a real value for the procedure. The conversion in general is defined by

$$\llbracket \theta_0 \rightarrow \theta_1 \vdash \theta'_0 \rightarrow \theta'_1 \rrbracket f = \llbracket \theta'_0 \vdash \theta_0 \rrbracket ; f ; \llbracket \theta_1 \vdash \theta'_1 \rrbracket,$$

where  $f$  is a function from  $\llbracket \theta_0 \rrbracket$  to  $\llbracket \theta_1 \rrbracket$ .

Finally, it is convenient to introduce new phrase types **1** and **0** such that

$$\theta \vdash \mathbf{1} \text{ and } \mathbf{0} \vdash \theta$$

for *every* phrase type  $\theta$ . As a result, **1** is a *greatest* type and **0** is a *least* type in the pre-order of phrase types.

We allow *every* phrase to have type **1** by adding the syntax rule

$$\overline{X : \mathbf{1}}$$

so that any phrase  $X$  has (at least) type **1** in *every* phrase-type assignment. But only phrases that are intuitively type-incorrect in a phrase-type assignment will have **1** as their *least* type in that phrase-type assignment. Phrase type **1** can be interpreted as denoting a singleton domain  $\{\perp\}$ ; then  $\llbracket \theta \vdash \mathbf{1} \rrbracket$  would be the unique function from  $\llbracket \theta \rrbracket$  to  $\llbracket \mathbf{1} \rrbracket$ , and

$$\text{cond}_1(e, m_0, m_1) = \perp.$$

A complete program whose least type is **1** should be considered as erroneous, and need not be executed because it has a trivial meaning. But, in order that errors can be localized, we would expect a compiler to generate a message for any phrase whose least type is **1** when none of its sub-phrases have least type **1**; however, this should be only a *warning* message, because, in general, a phrase need not have least type **1** when a proper component does. For example,

$$\text{let } x \text{ be } P \text{ in skip}$$

has type **comm**, even when the least type of  $P$  is **1**.

The least type  $\mathbf{0}$  and the coercion  $\mathbf{0} \vdash \theta$  are interpreted as follows:

$$\begin{aligned} \llbracket \mathbf{0} \rrbracket &= \{\perp\} \\ \llbracket \mathbf{0} \vdash \theta \rrbracket z &= \perp_\theta \end{aligned}$$

Type  $\mathbf{0}$  is not particularly useful, but we can replace the family of constants  $\mathbf{undef}[\theta]$  by the following single constant:

*Undefined*:

$$\overline{\mathbf{undef}: \mathbf{0}}$$

because its value ( $\perp$ ) converts to the least element of any phrase type.

This completes the description of the coercion system for our language. Our aim now is to demonstrate that the design is satisfactory by proving that the syntax and semantics have appropriate properties.

The first step is to demonstrate that there is a syntactic valuation  $\langle \cdot \rangle$  on phrases such that, for any phrase  $X$  and phrase-type assignment  $\pi$ ,  $\langle X \rangle \pi$  is a *least* type of  $X$  in  $\pi$ . This ensures that the language with coercions has most of the desirable attributes of explicitly typed languages.

Some of the equations defining  $\langle \cdot \rangle$  are as follows:

$$\begin{aligned} \langle 0 \rangle \pi &= \mathbf{val}[\mathbf{nat}] \\ \langle \iota \rangle \pi &= \begin{cases} \pi(\iota), & \text{if } \iota \in \text{dom } \pi \\ \mathbf{1}, & \text{otherwise;} \end{cases} \\ \langle \lambda \iota: \theta. P \rangle \pi &= \theta \rightarrow (\langle P \rangle (\pi \mid \iota \mapsto \theta)) \\ \langle PQ \rangle \pi &= \begin{cases} \theta', & \text{if } \langle P \rangle \pi = \theta \rightarrow \theta' \text{ and } \langle Q \rangle \pi \vdash \theta, \\ \mathbf{1}, & \text{otherwise;} \end{cases} \\ \langle \text{let } \iota \text{ be } P \text{ in } Q \rangle \pi &= \langle Q \rangle (\pi \mid \iota \mapsto \langle P \rangle \pi) \\ \langle V := E \rangle \pi &= \begin{cases} \mathbf{comm}, & \text{if } \langle V \rangle \pi = \mathbf{var}[\tau] \text{ and } \langle E \rangle \pi \vdash \mathbf{exp}[\tau], \\ \mathbf{1}, & \text{otherwise;} \end{cases} \\ \langle \mathbf{not } B \rangle \pi &= \begin{cases} \mathbf{val}[\mathbf{bool}], & \text{if } \langle B \rangle \pi \vdash \mathbf{val}[\mathbf{bool}], \\ \mathbf{exp}[\mathbf{bool}], & \text{if } \langle B \rangle \pi \vdash \mathbf{val}[\mathbf{bool}] \text{ fails to hold} \\ & \text{but } \langle B \rangle \pi \vdash \mathbf{exp}[\mathbf{bool}], \\ \mathbf{1}, & \text{otherwise;} \end{cases} \end{aligned}$$

Most of the remaining equations would be similar, but *conditional* phrases require a detailed treatment.

Suppose  $\langle B \rangle \pi \vdash \mathbf{val}[\mathbf{bool}]$ ; then the least type of **if**  $B$  **then**  $X_0$  **else**  $X_1$  in  $\pi$  should be a *least upper bound* of  $\langle X_0 \rangle \pi$  and  $\langle X_1 \rangle \pi$ , and similarly if  $\langle B \rangle \pi \vdash \mathbf{val}[\mathbf{bool}]$  fails to hold but  $\langle B \rangle \pi \vdash \mathbf{exp}[\mathbf{bool}]$ , except that if the least upper bound is a value-like type ( $\mathbf{val}[\tau]$  or of the form  $\theta \rightarrow \theta'$  for  $\theta'$  value-like), it must be 'raised' to the corresponding expression-like type (respectively,  $\mathbf{exp}[\tau]$  or  $\theta \rightarrow \theta''$ , where  $\theta''$  is the expression-like type corresponding to  $\theta'$ ). If  $\langle B \rangle \pi \vdash \mathbf{exp}[\mathbf{bool}]$  fails to hold, then the least type of **if**  $B$  **then**  $X_0$  **else**  $X_1$  in  $\pi$  is  $\mathbf{1}$ .



This means that, for any pair of types  $\theta$  and  $\theta'$ , there must be a *least* upper bound  $\theta \sqcup \theta'$  in the phrase-type pre-order. For example, the least upper bound of **val**[int] and **var**[real] is **exp**[real]. For the non-procedural types, the existence of a least upper bound  $\theta \sqcup \theta'$  for any pair of types  $\theta$  and  $\theta'$  is easily checked. For procedural types,

$$(\theta_0 \rightarrow \theta'_0) \sqcup (\theta_1 \rightarrow \theta'_1) = (\theta_0 \sqcap \theta_1) \rightarrow (\theta'_0 \sqcup \theta'_1)$$

and so we *also* need the existence of a greatest lower bound  $\theta \sqcap \theta'$  for any pair of types  $\theta$  and  $\theta'$ ; but this is also true for our system of types, and in particular,

$$(\theta_0 \rightarrow \theta'_0) \sqcap (\theta_1 \rightarrow \theta'_1) = (\theta_0 \sqcup \theta_1) \rightarrow (\theta'_0 \sqcap \theta'_1).$$

We now verify that valuation  $\langle \cdot \rangle$  has the desired properties; i.e., that any phrase has its least type as one of its types, and that the least type of a phrase does coerce into any type that the phrase has.

**Proposition 6.1.1.** *For any phrase  $X$  and phrase-type assignment  $\pi$ ,*

- (a)  $\pi \vdash X: \langle X \rangle \pi$ ;
- (b) *if  $\pi \vdash X: \theta$  then  $\langle X \rangle \pi \vdash \theta$ .*

Each part can be proved by structural induction.

It is evident that, for any  $\iota \in (\text{dom } \pi' \cap \text{dom } \pi)$ , if  $\pi'(\iota) \vdash \pi(\iota)$  then  $\langle \iota \rangle \pi' \vdash \langle \iota \rangle \pi$ ; remarkably, a property of this kind can be proved for *all* phrases in our language. In order to state the general result compactly, we re-define  $\pi' \vdash \pi$  as follows:

$$\pi' \vdash \pi \text{ iff } \text{dom } \pi \subseteq \text{dom } \pi' \text{ and, for all } \iota \in \text{dom } \pi, \pi'(\iota) \vdash \pi(\iota).$$

It is easily verified that  $\pi' \vdash X: \theta$  whenever  $\pi' \vdash \pi$  and  $\pi \vdash X: \theta$ . Notice that, when the pre-order on phrase types is discrete ( $\theta' \vdash \theta$  only if  $\theta' = \theta$ ), the definition of  $\pi' \vdash \pi$  reduces to the definition given in Section 3.1, and that  $\vdash$ , viewed as a binary relation on phrase-type assignments, is a pre-order.

**Proposition 6.1.2 (Monotonicity of  $\langle \cdot \rangle$ ).** *For any phrase  $X$  and phrase-type assignments  $\pi'$  and  $\pi$ , if  $\pi' \vdash \pi$  then  $\langle X \rangle \pi' \vdash \langle X \rangle \pi$ .*

**Proof.** By structural induction; we present two cases.

Case  $X = PQ$ . Assume  $\langle PQ \rangle \pi = \theta_1$ , with  $\langle P \rangle \pi = \theta_0 \rightarrow \theta_1$  and  $\langle Q \rangle \pi \vdash \theta_0$ .

By induction,  $\langle P \rangle \pi' = \theta'_0 \rightarrow \theta'_1$  with  $\theta'_0 \rightarrow \theta'_1 \vdash \theta_0 \rightarrow \theta_1$ , and

$$\langle Q \rangle \pi' \vdash \langle Q \rangle \pi \vdash \theta_0 \vdash \theta'_0$$

and so  $\langle PQ \rangle \pi' = \theta'_1 \vdash \theta_1$ .

Case  $X = \lambda\iota: \theta. P$ . Assume  $\langle \lambda\iota: \theta. P \rangle \pi = \theta \rightarrow \langle P \rangle (\pi \mid \iota \mapsto \theta)$ ;  $\pi' \vdash \pi$  implies  $(\pi' \mid \iota \mapsto \theta) \vdash (\pi \mid \iota \mapsto \theta)$  so that, by induction,

$$\langle P \rangle (\pi' \mid \iota \mapsto \theta) \vdash \langle P \rangle (\pi \mid \iota \mapsto \theta)$$

and so

$$\langle \lambda\iota: \theta. P \rangle \pi' = \theta \rightarrow \langle P \rangle (\pi' \mid \iota \mapsto \theta) \vdash \theta \rightarrow \langle P \rangle (\pi \mid \iota \mapsto \theta).$$

■

Our final task is to ensure that no combination of coercions and generic constructions is semantically ambiguous. We do this by temporarily considering a semantic interpretation  $\llbracket \cdot \rrbracket_{\pi \vdash X: \theta}: \llbracket \pi \rrbracket \rightarrow \llbracket \theta \rrbracket$  of *proofs* of sequents  $\pi \vdash X: \theta$ , defined by induction on the size of derivations (rather than on the syntactic structure of phrases). For example, the semantic equation corresponding to the Coercion rule

$$\frac{X: \theta}{X: \theta'} \text{ (when } \theta \vdash \theta')$$

is as follows:

$$\left[ \left[ \Psi \left\{ \frac{\Phi \left\{ \begin{smallmatrix} \vdots \\ \pi \vdash X: \theta \end{smallmatrix} \right\}}{\pi \vdash X: \theta'} \right\} \right] \right]_{\pi \vdash X: \theta'} = \left[ \left[ \Phi \left\{ \begin{smallmatrix} \vdots \\ \pi \vdash X: \theta \end{smallmatrix} \right\} \right] \right]_{\pi \vdash X: \theta} ; \llbracket \theta \vdash \theta' \rrbracket$$

The interpretation of proof  $\Psi$  is defined in terms of the interpretation of its sub-proof  $\Phi$ .

We can now state the following generalization of Lemma 3.4.2.

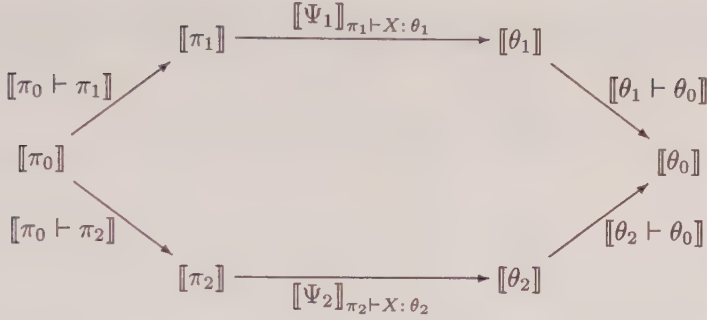
**Proposition 6.1.3.** *If  $\pi_0 \vdash \pi_1$ ,  $\pi_0 \vdash \pi_2$ ,  $\theta_1 \vdash \theta_0$ ,  $\theta_2 \vdash \theta_0$ , and  $\Psi_1$  and  $\Psi_2$  are proofs of  $\pi_1 \vdash X: \theta_1$  and  $\pi_2 \vdash X: \theta_2$ , respectively, then*

$$\llbracket \pi_0 \vdash \pi_1 \rrbracket ; \llbracket \Psi_1 \rrbracket_{\pi_1 \vdash X: \theta_1} ; \llbracket \theta_1 \vdash \theta_0 \rrbracket = \llbracket \pi_0 \vdash \pi_1 \rrbracket ; \llbracket \Psi_2 \rrbracket_{\pi_2 \vdash X: \theta_2} ; \llbracket \theta_2 \vdash \theta_0 \rrbracket,$$

where  $\llbracket \pi_i \vdash \pi_j \rrbracket: \llbracket \pi_i \rrbracket \rightarrow \llbracket \pi_j \rrbracket$  is defined component-wise; i.e.,

$$\llbracket \pi_i \vdash \pi_j \rrbracket u \iota = \llbracket \pi_i(\iota) \vdash \pi_j(\iota) \rrbracket (u(\iota))$$

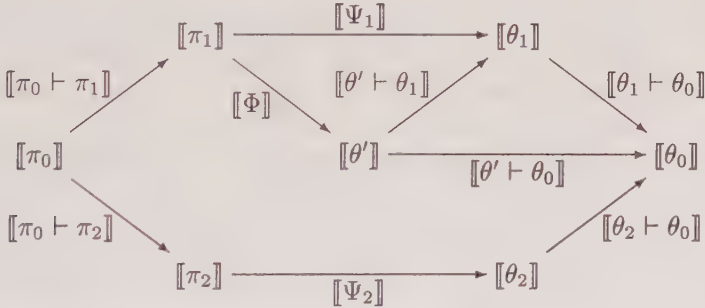
for every  $u \in \llbracket \pi_i \rrbracket$  and  $\iota \in \text{dom } \pi_j$ . The Proposition states that the following diagram commutes:



When  $\pi_0 = \pi_1 = \pi_2$  and  $\theta_0 = \theta_1 = \theta_2$ , the Proposition shows that any two proofs of a sequent yield the same semantics, and so the syntactic ambiguities do not create semantic ambiguities.

**Proof.** By induction on the sum of the sizes of proofs  $\Psi_1$  and  $\Psi_2$ .

We will discuss the case that the last step of  $\Psi_1$  is an instance of the Coercion rule. We then have:



for a proof  $\Phi$  of  $\pi_1 \vdash X: \theta'$  with  $\theta' \vdash \theta_1$ . The upper triangle commutes by the semantics of the Coercion rule, the right-hand triangle commutes by assumptions about the interpretations of coercions, and the lower part of the hexagon commutes by induction; hence, the outer hexagon commutes. ■

The case  $X = (E_0 = E_1)$  is noteworthy when both operands are of type **nat** because it is indeterminate whether the equality operation on natural, integer, or real numbers is used, and similarly if both operands are of type **int**; but there is no semantic ambiguity because the corresponding numerical conversions are *injective* (one-to-one) functions.

Similarly, we could add a generic addition operation to the language as follows:

*Addition:*

$$\frac{E_1: \text{val}[\tau] \quad E_2: \text{val}[\tau]}{E_1 + E_2: \text{val}[\tau]}$$

where  $\mathbf{nat} \vdash \tau \vdash \mathbf{real}$ ; to prevent ambiguity, the following diagram must commute whenever  $\mathbf{nat} \vdash \tau' \vdash \tau \vdash \mathbf{real}$ :

$$\begin{array}{ccccc} \llbracket \tau' \rrbracket & \times & \llbracket \tau' \rrbracket & \xrightarrow{\llbracket + \rrbracket_{\tau'}} & \llbracket \tau' \rrbracket \\ \downarrow & & \downarrow & & \downarrow \\ \llbracket \tau \rrbracket & \times & \llbracket \tau \rrbracket & \xrightarrow{\llbracket + \rrbracket_{\tau}} & \llbracket \tau \rrbracket \end{array}$$

where the horizontal arrows are the meanings of the addition operator for data types  $\tau'$  and  $\tau$ , and each of the vertical arrows is the conversion  $\llbracket \tau' \vdash \tau \rrbracket$ . This condition is satisfied by the usual definitions of addition on natural numbers, integers, and reals (disregarding overflow and round-off errors).

We can also allow the operator  $+$  to be used on numerical *expressions* because we have the following kind of commutativity:

$$\begin{array}{ccccc} \llbracket \mathbf{val}[\tau] \rrbracket & \times & \llbracket \mathbf{val}[\tau] \rrbracket & \xrightarrow{\llbracket + \rrbracket_{\mathbf{val}[\tau]}} & \llbracket \mathbf{val}[\tau] \rrbracket \\ \downarrow & & \downarrow & & \downarrow \\ \llbracket \mathbf{exp}[\tau] \rrbracket & \times & \llbracket \mathbf{exp}[\tau] \rrbracket & \xrightarrow{\llbracket + \rrbracket_{\mathbf{exp}[\tau]}} & \llbracket \mathbf{exp}[\tau] \rrbracket \end{array}$$

where the vertical arrows are conversions, and similarly for all of the other operators that might be used on both expressions and value phrases, such as  $=$ , **not**, **and**,  $<$ , **succ**, etc.

In summary, our language demonstrates that coercions are not *necessarily* undesirable, provided that the appropriate syntactic and semantic properties are rigorously verified to ensure non-ambiguity.

## 6.2 Local variables

Instead of providing all of the variables once and for all in the initial environment, programming languages since ALGOL 60 have allowed for *dynamically created* variables; i.e., provided a way for a programmer to specify the creation of 'new' variables during program execution, and, sometimes, a way to dispose of such variables. In fact, ALGOL 60 provided only a very structured form of dynamic storage, the *locally declared* variable, whose lifetime is determined by the scope of the identifier that denotes it. This allows for more efficient use of memory and more control of interference between program parts than fully static storage, but is much simpler to use and implement than fully dynamic storage.

To provide this kind of facility in our example language, we introduce a family of quantifier constants

$$\mathbf{new}[\tau]: (\mathbf{var}[\tau] \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$$

so that

$$\# \mathbf{new}[\tau] \iota. C$$

is a variable-declaration block with command  $C$  as its body. A new variable is created for each execution of such a block, and that variable can be de-allocated by the implementation when that execution is finished. For example, the simple **for** loop described in Section 2.1 can now be defined by the following equivalence:

$$\mathbf{for } N \mathbf{ do } C \equiv \# \mathbf{new}[\mathbf{nat}] \iota. \iota := N; \mathbf{while } \iota \neq 0 \mathbf{ do } \iota := \iota - 1; C$$

where  $\iota$  is not free in  $N$  or  $C$ ; more complex loop forms can be similarly treated.

In the rest of this section, we briefly describe the technique that has traditionally been used in denotational semantics to treat dynamic storage. The essential idea is that components of the state are indexed, not by identifiers, but by special values, termed *locations*, which can be thought of as abstract storage addresses. Allocation and de-allocation of variables can then be treated by recording in the state itself whether or not each component is currently ‘in use’. If locations themselves are storable values, it is possible to describe the semantics of features such as the ‘pointer’ types in PASCAL that permit creation and use of dynamically-linked structures.

This suggests the following definition of the set of states:

$$S = L \rightarrow (V + \{\mathit{unused}\}),$$

where  $L$  is a suitable set of locations (e.g., the set of natural numbers),  $V$  is the set of storable values, and  $+$  denotes disjoint union of sets. For simplicity, in this section we shall assume that there is a single set  $V$  containing all of the storable values, and that  $\mathit{unused} \notin V$ , so that ‘tags’ are unnecessary in forming elements of the disjoint union. The following functions would then allow storage components to be allocated and de-allocated, and the values of the components to be accessed and updated:

$$\begin{aligned} \mathit{new}: S &\rightarrow L_{\perp} \\ \mathit{lose}: L &\rightarrow \llbracket \mathbf{comm} \rrbracket \\ \mathit{contents}: L &\rightarrow (S \rightarrow V_{\perp}) \\ \mathit{update}: L &\rightarrow (V \rightarrow \llbracket \mathbf{comm} \rrbracket) \end{aligned}$$

The first of these can be any ‘choice’ function such that, for any  $s \in S$ ,  $s(\mathit{new}(s)) = \mathit{unused}$  if there exists some  $l \in L$  such that  $s(l) = \mathit{unused}$ , and  $\mathit{new}(s) = \perp$  otherwise. The remaining functions are defined as follows:



$$\begin{aligned}
lose(l)(s) &= (s \mid l \mapsto \text{unused}) \\
contents(l)(s) &= \begin{cases} v, & \text{if } s(l) = v \in V; \\ \perp, & \text{if } s(l) = \text{unused}. \end{cases} \\
update(l)(v)(s) &= (s \mid l \mapsto v)
\end{aligned}$$

This approach to the semantics of dynamic storage corresponds to the way implementations deal with fully-dynamic storage, but not to the conventional implementation of local-variable declarations. The unsatisfactory nature of the semantic interpretation can be seen by trying to show the equivalence of

$$\# \text{new}[\text{nat}] n.$$

$c$

and  $c$ , where  $c$  is a non-local identifier of type **comm**. Intuitively, the meaning of  $c$  cannot access the ‘new’ local variable  $n$ ; but, semantically,  $c$  can range over *arbitrary* command meanings, including those that test whether the location denoted by  $n$  is ‘unused’, and so the equivalence fails.

A more sophisticated approach to the semantics of local-variable declarations will be described in Section 7.4.

### 6.3 Product types and arrays

In this section, we add to our language a simple form of ‘structured’ type similar to the ‘classes’ of SIMULA, the ‘records’ of ALGOL W, PASCAL and ML, and the ‘modules’ of languages such as MODULA 2, and consider how *arrays* can be treated.

The first step is to introduce a new kind of phrase-type expression, as follows:

$$\theta ::= \dots \mid \iota_1 \mapsto \theta_1 \ \& \dots \& \iota_n \mapsto \theta_n \quad (\iota_i \text{ distinct})$$

The new types are termed *product* types; notice that they are *phrase* types, rather than *data* types. Each of the *field types*  $\theta_i$  can be any of the (phrase) types, and each of the *field names*  $\iota_i$  can be any identifier (different from the other field names of that product type). Semantically,

$$\llbracket \iota_1 \mapsto \theta_1 \ \& \dots \& \iota_n \mapsto \theta_n \rrbracket = \prod_{i=1}^n \llbracket \theta_i \rrbracket,$$

that is, a product type denotes the set of all  $n$ -tuples of meanings of appropriate type, ordered component-wise.

We now consider what coercions are to be allowed on products. We want product types to be monotonic with respect to their component types:

$$\iota_1 \mapsto \theta_1 \ \& \dots \& \iota_n \mapsto \theta_n \vdash \iota_1 \mapsto \theta'_1 \ \& \dots \& \iota_n \mapsto \theta'_n$$

if, for all  $1 \leq i \leq n$ ,  $\theta_i \vdash \theta'_i$ . But we also want to allow components that are unnecessary in some context to be ‘dropped’; the general coercion on product types should then be as follows:

$$\iota_1 \mapsto \theta_1 \ \& \ \dots \ \& \ \iota_n \mapsto \theta_n \vdash \iota'_1 \mapsto \theta'_1 \ \& \ \dots \ \& \ \iota'_m \mapsto \theta'_m$$

if  $\{\iota'_1, \dots, \iota'_m\} \subseteq \{\iota_1, \dots, \iota_n\}$  and  $\theta_i \vdash \theta'_j$  whenever  $\iota'_j = \iota_i$ . Notice that, as a binary relation on phrase-type expressions,  $\vdash$  is now a pre-order that is *not* a partial order: product types that differ only by a permutation of fields are  $\simeq$ -equivalent but not equal. The conversion corresponding to the coercion is defined as follows:

$$\begin{aligned} & \llbracket \iota_1 \mapsto \theta_1 \ \& \ \dots \ \& \ \iota_n \mapsto \theta_n \vdash \iota'_1 \mapsto \theta'_1 \ \& \ \dots \ \& \ \iota'_m \mapsto \theta'_m \rrbracket (p)(j) \\ & = \llbracket \theta_i \vdash \theta'_j \rrbracket (p(i)) \end{aligned}$$

when  $\iota'_j = \iota_i$ .

To allow meanings of product types to be expressed and used, we add the following syntax rules:

*Product introduction:*

$$\frac{X_i: \theta_i \quad \text{for every } i = 1, \dots, n}{\iota_1 \mapsto X_1, \dots, \iota_n \mapsto X_n: \iota_1 \mapsto \theta_1 \ \& \ \dots \ \& \ \iota_n \mapsto \theta_n} \quad (\iota_i \text{ distinct})$$

*Field selection:*

$$\frac{P: \iota \mapsto \theta}{P.\iota: \theta}$$

The rule for Field selection can be particularly simple because only the field being selected need be mentioned explicitly; the other fields are simply dropped by the coercion on product types. The occurrences of  $\iota$  as a field name in  $\dots, \iota \mapsto X, \dots$  and  $P.\iota$  are not conventional identifier occurrences; for example, they are not subject to substitutions.

We present an example of how these facilities could be used for doing ‘object-oriented’ programming. The program fragment in Table 10 illustrates how to define a class of ‘counter’ objects, whose capabilities are limited to incrementation and evaluation, and create an instance of the class.

```

let counter(user: (inc ↦ comm & val ↦ exp[nat]) → comm) =
  # new[nat] n. n := 0;
    user(inc ↦ n := n + 1, val ↦ n)
in
  ⋮
  ( # counter c.
    ⋯ c.inc ; ⋯ c.val ⋯ )
  ⋮

```

**Table 10.** A class of counter objects

The procedural parameter *user* may be thought of as a typical ‘customer’ for an instance of the counter class. Note that the representation of a counter (i.e., variable *n*) is a ‘private’ variable, not directly accessible to users. In more complex examples, private *procedures* might also be necessary.

Semantically, we define

$$\llbracket \iota_1 \mapsto X_1, \dots, \iota_n \mapsto X_n \rrbracket(u)(i) = \llbracket X_i \rrbracket u,$$

for  $1 \leq i \leq n$ ,

$$\llbracket P.\iota \rrbracket_{\pi\theta}(u) = \llbracket P \rrbracket_{\pi(\iota \mapsto \theta)}(u)(1)$$

and

$$\text{cond}_{\iota_1 \mapsto \theta_1 \& \dots \& \iota_n \mapsto \theta_n}(b, p_1, p_2)(i) = \text{cond}_{\theta_i}(b, p_1(i), p_2(i))$$

for  $1 \leq i \leq n$ , provided  $\text{cond}_{\theta_i}$  is definable for each of the  $\theta_i$ .

The following equivalences can now be validated:

- for any  $X_i: \theta_i$  ( $1 \leq i \leq n$ ),

$$(\iota_1 \mapsto X_1, \dots, \iota_n \mapsto X_n).\iota_i \equiv_{\theta_i} X_i;$$

- for  $\theta = \iota_1 \mapsto \theta_1 \& \dots \& \iota_n \mapsto \theta_n$  and any  $P: \theta$ ,

$$(\iota_1 \mapsto P.\iota_1, \dots, \iota_n \mapsto P.\iota_n) \equiv_{\theta} P;$$

- for any  $B: \text{exp}[\text{bool}]$  and  $P_1, P_2: \iota \mapsto \theta$ ,

$$(\text{if } B \text{ then } P_1 \text{ else } P_2).\iota \equiv_{\theta} \text{if } B \text{ then } P_1.\iota \text{ else } P_2.\iota,$$

and similarly if  $B: \text{val}[\text{bool}]$ .

Arrays differ from products in two ways: components of arrays can be selected by using a *computed* ‘index’ (rather than an explicit field name), and, to allow a practical implementation and type checking, all of the components of an array must have the same type. Programming languages

typically provide arrays in two specialized and unrelated forms: as conventional arrays of variables and in ‘case’ constructions, which involve indexing into what are essentially arrays of commands or expressions. Furthermore, the only substantive difference between conventional arrays and procedures with **var** result types is representation. We can unify and generalize all of these possibilities by simply regarding arrays as procedures, rather than introducing specialized types into the language.

```

let NewRealVarArray
  (size: exp[nat],
   user: (exp[nat] → var[real]) → comm) =
  letrec allocate(i: exp[nat], a: exp[nat] → var[real]): comm =
    if i = size then user(a) else
      # new[real] x.
        let newa(j: exp[nat]) = if i = j then x else a(j)
        in allocate(succ i, newa)
  in allocate(0, undef[exp[nat] → var[real]])
in ...

```

Table 11. An array declarator

For example, a programmer can, in principle, define a procedure *NewRealVarArray* as in Table 11; then,

$$\# \text{NewRealVarArray}(n)A. \dots A(i) \dots$$

declares *A* to be an ‘array’ of *n* new real-valued variables. The allowable ‘subscripts’ are expressions whose values are natural numbers less than *n*.

The implementation of *NewRealVarArray* works as follows. Each call of *allocate* (except the last) declares one real variable, passing on a procedure that allows access to this new variable and also to previously allocated ones. The last call of *allocate* (when *i* reaches *size*) applies the *user* procedure to the array of variables, represented by the procedure. This technique is possible even in languages such as PASCAL that are supposed not to have dynamic arrays. In practice, equivalent but more efficient implementations of *NewRealVarArray* and similar quantifiers should be provided as the values of constants or pre-defined identifiers.

**Exercise 6.3.1.** Suppose that we add the following new construct to our language:

*Case selection:*

$$\frac{N: \mathbf{exp}[\mathbf{nat}] \quad F: \mathbf{exp}[\mathbf{nat}] \rightarrow \theta}{\mathbf{case} \ N \ \mathbf{of} \ F: \theta}$$

and interpret it by the following equivalence:

$$\text{case } N \text{ of } F \equiv_{\theta} F(N);$$

that is, the **case** construction is an alternative notation for a procedure call. For this approach to **case** selection to be useful, it is necessary to add new notation for defining procedures by explicit enumeration of argument-to-result associations. Design such notation, including a specification of the syntax and semantics.

## 6.4 Lists

Functional languages usually allow data to be organized into *lists* of arbitrary (even infinite) length. Consider adding a new class of phrase types as follows to our Algol-like language:

$$\theta ::= \dots \mid \mathbf{list}[\theta]$$

We allow the following ways of defining lists:

*Empty list:*

$$\overline{\mathbf{nil}: \mathbf{list}[\mathbf{0}]}$$

*Prefixed list:*

$$\frac{X: \theta \quad L: \mathbf{list}[\theta]}{X :: L: \mathbf{list}[\theta]}$$

If we adopt the coercion  $\mathbf{list}[\theta] \vdash \mathbf{list}[\theta']$  whenever  $\theta \vdash \theta'$ , the constant **nil** coerces into the empty list of *any* list type. The list  $X :: L$  has  $X$  as its first component and  $L$  as the list of remaining components. Finally, we provide the following construct to allow lists to be tested for emptiness and decomposed:

*List analysis:*

$$\frac{\begin{array}{c} \left[ \begin{array}{l} \iota_1: \theta \\ \iota_2: \mathbf{list}[\theta] \end{array} \right] \\ \vdots \\ L: \mathbf{list}[\theta] \quad X: \theta' \quad X': \theta' \end{array}}{\mathbf{ifnull } L \text{ then } X \text{ else } (\iota_1 :: \iota_2). X': \theta'}$$

If the list  $L$  is empty, the result of evaluating this construct is  $X$ ; otherwise, the result is  $X'$  with  $\iota_1$  and  $\iota_2$  bound to the first component and the rest of list  $L$ , respectively.

As an example of the use of these facilities, the following defines a function that applies its first argument to every component of its second argument:



**letrec**  $map(f: \theta \rightarrow \theta', l: \text{list}[\theta]): \text{list}[\theta'] =$   
     **ifnull**  $l$  **then** **nil** **else**  $(hd :: tl). f(hd) :: map(f, tl)$   
**in**  $\dots$

Here  $\theta$  and  $\theta'$  are arbitrary phrase types.

To define the semantics of these facilities, we need a domain  $\llbracket \text{list}[\theta] \rrbracket$  that satisfies the following domain isomorphism:

$$D \cong (\{\text{nil}\} + \llbracket \theta \rrbracket \times D)_{\perp}$$

where the domain  $D_0 + D_1$  for domains  $D_0$  and  $D_1$  is the disjoint union of their underlying sets, with the ordering inherited from  $D_0$  and  $D_1$ . Here *nil* is the empty list, and the use of ‘lifting’ gives the domain a least element, so that lists can be recursively defined. We defer discussion on the existence of solutions to such domain equations.

Assuming a suitable such domain  $\llbracket \text{list}[\theta] \rrbracket$ , the semantic equations are as follows (with tags and isomorphisms omitted):

$$\begin{aligned} \llbracket \text{nil} \rrbracket u &= \text{nil} \\ \llbracket X :: L \rrbracket u &= (\llbracket X \rrbracket u, \llbracket L \rrbracket u) \\ \llbracket \text{ifnull } L \text{ then } X \text{ else } (\iota_1 :: \iota_2). X' \rrbracket u \\ &= \begin{cases} \llbracket X \rrbracket u, & \text{if } \llbracket L \rrbracket u = \text{nil} \\ \llbracket X' \rrbracket (u \mid \iota_1 \mapsto v \mid \iota_2 \mapsto l), & \text{if } \llbracket L \rrbracket u = (v, l) \\ \perp, & \text{if } \llbracket L \rrbracket u = \perp \end{cases} \end{aligned}$$

Note that list prefixing is ‘lazy’, in the sense that it is possible to define lists that are conceptually of infinite length; for example,

**letrec** *zeroes*:  $\text{list}[\text{val}[\text{nat}]]$  **be**  $0 :: \text{zeroes}$  **in**  $\dots$

defines what can be thought of as an *infinite* list of zeroes. In fact, an implementation would create a finite structure with a ‘loop’. Similarly,

**letrec** *numbers*:  $\text{list}[\text{val}[\text{nat}]]$  **be**  $0 :: map(\text{succ}, \text{numbers})$  **in**  $\dots$

defines the list of all natural numbers when  $\text{succ} = \lambda n: \text{val}[\text{nat}]. n + 1$  and  $map$  is as defined above; in this case, an implementation would only need to create as many components as are actually used by the rest of the program.

Finally, the conversion  $\llbracket \text{list}[\theta] \vdash \text{list}[\theta'] \rrbracket$  can be defined as follows: the least element  $\perp$  and the empty list *nil* are preserved, and a pair  $(v, l)$  for  $v \in \llbracket \theta \rrbracket$  and  $l \in \llbracket \text{list}[\theta] \rrbracket$  is converted by applying  $\llbracket \theta \vdash \theta' \rrbracket$  to  $v$  and, recursively,  $\llbracket \text{list}[\theta] \vdash \text{list}[\theta'] \rrbracket$  to  $l$ .

We now consider the question of how to solve domain equations of the form

$$D \cong \dots D \dots$$

Consider first the following example:

$$D \cong \{nil\} + T \times D,$$

where  $T = \{true, false\}$ ; this is simpler than the equation for  $\llbracket \text{list}[\theta] \rrbracket$  because there is no lifting (and  $T$  does not have a least element). We proceed by analogy with the method used to solve equations of the form

$$c = \cdots c \cdots$$

in Section 2.3; i.e., start with an initial object  $c_0$ , define better-and-better approximations by the inductive rule  $c_{i+1} = \cdots c_i \cdots$ , and obtain the desired 'least' solution as the limit (union) of all of the  $c_i$ . Here, we let  $D_0 = \emptyset$  (the empty set) and define

$$D_{i+1} \equiv \{nil\} + T \times D_i$$

for every  $i \in \omega$ . This gives us

- $D_0 = \emptyset$
- $D_1 = \{nil\} + T \times \emptyset = \{nil\} + \emptyset = \{(0, nil)\}$
- $D_2 = \{nil\} + T \times D_1$   
 $= \{(0, nil)\} \cup \{(1, (true, (0, nil))), (1, (false, (0, nil)))\}$
- $D_3 = \{nil\} + T \times D_2$   
 $= \{(0, nil)\} \cup \{(1, (true, (0, nil))), (1, (false, (0, nil)))\}$   
 $\cup \{(1, (true, (1, (true, (0, nil))))),$   
 $(1, (true, (1, (false, (0, nil))))),$   
 $(1, (false, (1, (true, (0, nil))))),$   
 $(1, (false, (1, (false, (0, nil))))), \}$

and so on. If we regard element  $(0, nil)$  as representing the empty sequence, and an element of the form  $(1, (t, s))$  as representing the non-empty sequence with first component  $t$  and remainder  $s$ , we see that, for each  $i \in \omega$ ,  $D_i$  is the set of truth-value sequences having length less than  $i$ .

Each  $D_i$  can be embedded in  $D_{i+j}$  for all  $j \in \omega$ . In fact,  $D_i \subseteq D_{i+j}$ , and so we can define the set

$$D_\infty = \bigcup_{i \in \omega} D_i$$

which is a set of (representations of) all finite sequences of truth values, and this set does satisfy the equation

$$D = \{nil\} + T \times D$$

when it is substituted for  $D$ . Of course, there is no reason to distinguish between isomorphic domains and so the set  $T^* = \sum_{i \in \omega} T^i$ , which is isomorphic to  $D_\infty$ , can also be regarded as a solution to the isomorphism.

There are also solutions that are *not* isomorphic to  $D_\infty$ ; for example, the set  $T^* \cup T^\omega$  of all finite and denumerably infinite truth-value sequences satisfies the isomorphism

$$D \cong \{nil\} + T \times D.$$

The special significance of  $D_\infty$  is that it is, up to isomorphism, the *least* such solution of the equation in a certain sense.

Consider now the domain equation

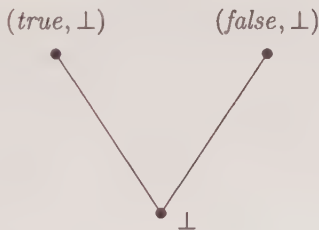
$$D \cong (T \times D)_\perp,$$

which allows for 'lifting' but not for an explicit empty list. We again start with  $D_0 = \emptyset$  and, for every  $i \in \omega$ , define

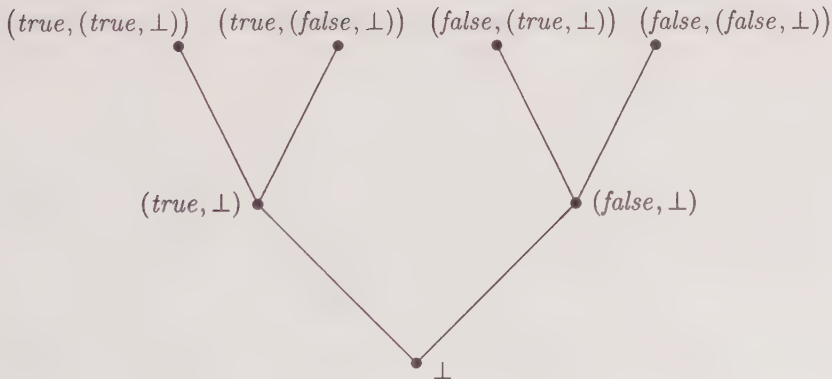
$$D_{i+1} = (T \times D_i)_\perp$$

this gives us

- $D_0 = \emptyset$
- $D_1 = (T \times D_0)_\perp = \{\perp\}$
- $D_2 = (T \times D_1)_\perp =$



- $D_3 = (T \times D_2)_\perp =$



and so on. If we regard  $\perp$  as representing the empty sequence and  $(t, s)$  as representing the non-empty sequence with first component  $t$  and remainder  $s$ , we see that each  $D_i$  is the domain of truth-value sequences having length less than  $i$ , ordered by the *prefix* ordering; i.e.,  $s \sqsubseteq s'$  if and only if  $s$  is a prefix (initial subsequence) of  $s'$ .

For every  $i$ ,  $D_i$  is, in a certain sense, a *sub-domain* of  $D_{i+j}$  for all  $j \in \omega$ . In fact, if we identify the least elements,  $D_i$  is a sub-poset of each of the  $D_{i+j}$ , and so we can define the union  $\bigcup_{i \in \omega} D_i$  and partially order it by the prefix relation, which extends the ordering in each of the  $D_i$ . This poset satisfies the equation

$$D \cong (T \times D)_{\perp};$$

however, it is not a satisfactory solution. The problem is that  $\bigcup_{i \in \omega} D_i$  is not a domain! Consider any  $\omega$ -sequence of truth values

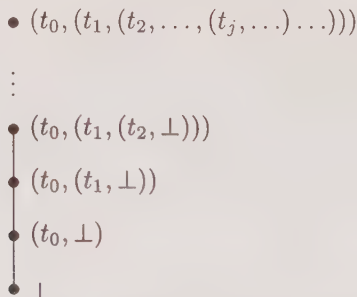
$$t_0, t_1, t_2, \dots, t_j, \dots;$$

the  $\omega$ -chain of its finite prefixes

$$\perp \sqsubseteq (t_0, \perp) \sqsubseteq (t_0, (t_1, \perp)) \sqsubseteq (t_0, (t_1, (t_2, \perp))) \sqsubseteq \dots$$

must have as its least upper bound a representation of the *infinite* sequence of truth values; but every element of the union represents a *finite* sequence, and so the union is not  $\omega$ -chain complete.

The union must be completed by adding a 'top layer' of limit points which are essentially denumerably infinite sequences of truth values. Each such limit point is approximated by all of its finite prefixes, as in the following Hasse-diagram fragment:



**Exercise 6.4.1.** Use the method described above to determine the first few approximating domains for the domain equation

$$D \cong (\{\text{nil}\} + [\![\theta]\!] \times D)_{\perp}$$

and describe the limit domain.

We will not attempt to give here a more rigorous or general explanation of the method for solving recursive domain equations; see the chapter on Domain Theory in this volume. It must be realized, however, that some such equations have *no* solutions. Consider, for example, the domain equation

$$D \cong B + (D \rightarrow D) .$$

If  $D \rightarrow D$  is the set of all functions on set  $D$ , this equation does not have non-trivial solutions. To see this, notice that  $d$ , the number of elements of  $D$ , must satisfy the equation  $d = b + d^d$ , where  $b$  is the cardinality of  $B$ , and the only possibility is  $d = 1$  when  $b = 0$ . To obtain a domain of values appropriate to this application, one must allow for possible *non-termination* of execution by either using partial functions, as in

$$D \cong B + (D \rightarrow D),$$

or ‘lifting’, as in

$$D \cong (B + (D \rightarrow D))_{\perp}.$$

By ensuring that  $D$  is a domain with a non-trivial ordering, the cardinality of the function spaces is reduced because the functions are required to be *continuous*, and the equations then have non-trivial solutions.

## 6.5 Acceptors

The treatment of variables in our Algol-like language, though typical of most existing languages, is somewhat asymmetric in that, for every data type  $\tau$ , phrases of type **var** $[\tau]$  can be used as both the left *and* right-hand sides of assignments, but phrases of type **exp** $[\tau]$  can be used as right-hand sides *only*. A more symmetric treatment of variables can be achieved by introducing ‘write-only variables’, phrases that can be used as left-hand sides of assignments, but not as right-hand sides. For example, if it is intended that the formal parameter of a procedure is to be used *only* as a ‘result’ parameter, this should be made explicit in the argument type of the procedure. Such write-only variables will be termed *acceptors*.

We therefore add new phrase types as follows:

$$\theta ::= \dots \mid \mathbf{acc}[\tau],$$

replace the syntax rule for assignments by the following rule:

*Assignment:*

$$\frac{A: \mathbf{acc}[\tau] \quad E: \mathbf{exp}[\tau]}{A := E: \mathbf{comm}}$$

and add a new coercion



$$\mathbf{var}[\tau] \vdash \mathbf{acc}[\tau]$$

for every data type  $\tau$  to allow conventional variables to be used as left-hand sides of assignments.

Semantically, it is now convenient to regard a variable abstractly as an ‘object’ consisting of two components that correspond to the two ways it can be used: on the right-hand side of assignments as an expression, and on the left-hand side of assignments as an acceptor which accepts an expression meaning and then behaves like a command. Sometimes, the acceptor and expression components of a variable meaning are termed its *l-value* and *r-value*, respectively, referring to the left and right-hand sides of the assignment command. The appropriate domain definitions are as follows:

$$\begin{aligned} \llbracket \mathbf{acc}[\tau] \rrbracket &= \llbracket \mathbf{exp}[\tau] \rrbracket \rightarrow \llbracket \mathbf{comm} \rrbracket \\ \llbracket \mathbf{var}[\tau] \rrbracket &= \llbracket \mathbf{acc}[\tau] \rrbracket \times \llbracket \mathbf{exp}[\tau] \rrbracket \end{aligned}$$

The semantic equation for assignment commands is then

$$\llbracket A := E \rrbracket u = \llbracket A \rrbracket u (\llbracket E \rrbracket u)$$

and the coercions for variables are interpreted as follows:

$$\begin{aligned} \llbracket \mathbf{var}[\tau] \vdash \mathbf{exp}[\tau] \rrbracket (a, e) &= e \\ \llbracket \mathbf{var}[\tau] \vdash \mathbf{acc}[\tau] \rrbracket (a, e) &= a \end{aligned}$$

The variables in the initial environment  $u_0$  for programs can be defined to access and update the ‘components’ of states as follows. For all  $\iota \in [\tau]$  for some  $\tau$ ,

$$\begin{aligned} u_0(\iota) &= (a_\iota, e_\iota) \\ \text{where } a_\iota(e)(s) &= \begin{cases} \text{undefined,} & \text{if } e(s) = \perp \\ (s \mid \iota \mapsto e(s)), & \text{otherwise} \end{cases} \\ \text{and } e_\iota(s) &= s(\iota). \end{aligned}$$

Finally, to allow for conditional acceptors and variables, we define

$$\mathit{cond}_{\mathbf{acc}[\tau]}(b, a_0, a_1)(e) = \mathit{cond}_{\mathbf{comm}}(b, a_0(e), a_1(e))$$

for all  $e \in \llbracket \mathbf{exp}[\tau] \rrbracket$ , and

$$\begin{aligned} \mathit{cond}_{\mathbf{var}[\tau]}(b, (a_0, e_0), (a_1, e_1)) &= (a, e) \\ \text{where } a &= \mathit{cond}_{\mathbf{acc}[\tau]}(b, a_0, a_1) \\ \text{and } e &= \mathit{cond}_{\mathbf{exp}[\tau]}(b, e_0, e_1) \end{aligned}$$

Note that it is consistent to regard  $\mathbf{acc}[\tau]$  as an *abbreviation* of the procedural type  $\mathbf{exp}[\tau] \rightarrow \mathbf{comm}$ , rather than as an additional primitive

type. This suggests that any data-type coercion  $\tau \vdash \tau'$  should also induce the following coercion on acceptor phrase types:

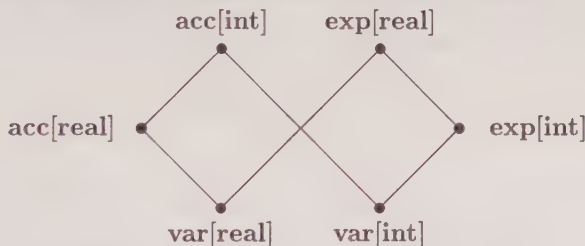
$$\mathbf{acc}[\tau'] \vdash \mathbf{acc}[\tau],$$

because  $\mathbf{exp}[\tau] \vdash \mathbf{exp}[\tau']$ . For example, an acceptor for real numbers should be coercible to an acceptor for integers; if the latter is applied to an integer-producing expression, the integer can be converted to a real number.

The addition of acceptor phrases as described above has, however, a subtle defect: if  $\tau_1$  and  $\tau_2$  are comparable but inequivalent data types, the phrase types  $\mathbf{var}[\tau_1]$  and  $\mathbf{var}[\tau_2]$  have upper bounds but do not have a *least* upper bound, and this means that the conditional phrase

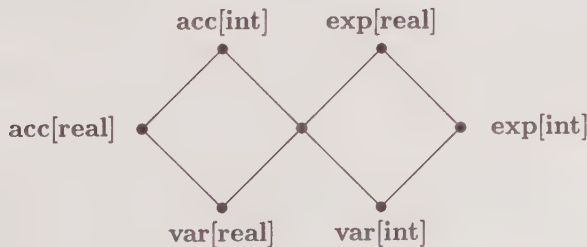
$$\mathbf{if } B \mathbf{ then } X_1 \mathbf{ else } X_2$$

is not well-formed if  $X_1: \mathbf{var}[\tau_1]$  and  $X_2: \mathbf{var}[\tau_2]$ . For example, if  $\tau_1 = \mathbf{real}$  and  $\tau_2 = \mathbf{int}$ , both  $\mathbf{acc}[\mathbf{int}]$  and  $\mathbf{exp}[\mathbf{real}]$  are common upper bounds of  $\mathbf{var}[\mathbf{real}]$  and  $\mathbf{var}[\mathbf{int}]$ , but there is no *least* upper bound:



The problem is that the conditional phrase is essentially a variable that can be used as an integer acceptor and as a real expression; however, our type system does not allow for such variables of 'mixed type'.

The solution to this problem is to introduce new types of the form  $\mathbf{acc}[\tau] \ \& \ \mathbf{exp}[\tau']$ , which would coerce *both* to  $\mathbf{acc}[\tau]$  and to  $\mathbf{exp}[\tau']$ , and would also be greater than any other lower bound. For example, the unlabelled point in the middle of the following Hasse diagram is the new type  $\mathbf{acc}[\mathbf{int}] \ \& \ \mathbf{exp}[\mathbf{real}]$ , and is now a *least* upper bound for  $\mathbf{var}[\mathbf{real}]$  and  $\mathbf{var}[\mathbf{int}]$ :



This allows **if**  $B$  **then**  $X_1$  **else**  $X_2$  when  $X_1: \text{var}[\text{real}]$  and  $X_2: \text{var}[\text{int}]$ . It is actually possible to define a binary operation  $\&$  applicable to *any* types, but we will only need types of the form  $\text{acc}[\tau] \& \text{exp}[\tau']$  here.

Semantically, we want  $\llbracket \text{acc}[\tau] \& \text{exp}[\tau'] \rrbracket = \llbracket \text{acc}[\tau] \rrbracket \times \llbracket \text{exp}[\tau'] \rrbracket$ ; the coercions from  $\text{acc}[\tau] \& \text{exp}[\tau']$  to  $\text{acc}[\tau]$  and  $\text{exp}[\tau']$  are the projection functions on  $\llbracket \text{acc}[\tau] \rrbracket \times \llbracket \text{exp}[\tau'] \rrbracket$ . We can now *define*  $\text{var}[\tau]$  to be an abbreviation for  $\text{acc}[\tau] \& \text{exp}[\tau]$  (rather than a primitive type); the coercions on the type  $\text{acc}[\tau] \& \text{exp}[\tau]$  are the coercions desired for  $\text{var}[\tau]$ .

## 6.6 Jumps

In this section, we introduce ‘jumps’ into our Algol-like language and discuss the technique of *continuations*, which makes it possible to define compositional semantics of such features.

### 6.6.1 Completions

The reader may recall that it was convenient in Section 2.4 to assume a program-completion function  $k_0: S \rightarrow O$  that mapped final states of program execution to program outputs. To allow other ways of ‘completing’ program execution, we introduce a new phrase type of *completions*:

$$\theta ::= \dots \mid \text{compl}$$

with

$$\llbracket \text{compl} \rrbracket = S \rightarrow O$$

As examples of completions, we introduce the following new syntax:

*Immediate termination:*

$$\overline{\text{stop: compl}}$$

*Abortion:*

$$\overline{\text{abort: compl}}$$

The intention is that a programmer can use **stop** or **abort** to immediately terminate program execution; **stop** does the usual program finalization (such as output), whereas **abort** produces an error message.

To specify the semantics of these, we partition the domain of outputs as follows:

$$O = A + \{\text{error}\}$$

where *error* represents the error message produced by **abort**, and  $A$  is a suitable domain of ‘answers’ (i.e., outputs that are *not* error messages).

If we assume  $error \notin A$ , tags are unnecessary in forming elements of the disjoint union, and then we can define

$$\begin{aligned} \llbracket \text{stop} \rrbracket_{us} &= k_0(s) \\ \llbracket \text{abort} \rrbracket_{us} &= error \end{aligned}$$

for any environment  $u$  and state  $s$ , where  $k_0: S \rightarrow A$  is the standard program-completion function.

However, we have not yet provided a way that such completions can actually be used by a programmer. In many languages, this is done by introducing a command form **goto**  $K$ , where  $K$  is a completion, but it is simpler to introduce the coercion  $\text{compl} \vdash \text{comm}$ , so that a completion can be used wherever a command is expected; the command resulting from the conversion ignores the ‘normal’ continuation of the computation, simply initiating execution of the completion.

To describe the semantics of jumps, we cannot continue to use

$$\llbracket \text{comm} \rrbracket = S \rightarrow S,$$

because a command containing a completion might produce a final program output and not an (intermediate) state. Hence, we re-define  $\llbracket \text{comm} \rrbracket$  to be  $\llbracket \text{compl} \rrbracket \rightarrow \llbracket \text{compl} \rrbracket$ . Expanding this out, we get

$$\llbracket \text{comm} \rrbracket = (S \rightarrow O) \rightarrow S \rightarrow O$$

so that, if  $c \in \llbracket \text{comm} \rrbracket$ ,  $c(k)(s)$  is a *program output*. The second argument,  $s \in S$ , is the state *before* execution of the command. The first argument,  $k \in (S \rightarrow O)$ , is termed the *continuation* for the command execution; the continuation specifies how the state *after* execution of the command (if the execution terminates without a jump) is to be mapped into the output for the whole program.

We must now re-define the semantics of all of the forms of command. This new ‘continuation semantics’ for commands is given in Table 12;  $k$  ranges over  $\llbracket \text{compl} \rrbracket$ .

$$\begin{aligned} \llbracket \text{skip} \rrbracket_{uks} &= k(s) \\ \llbracket C_0 ; C_1 \rrbracket_{uks} &= \llbracket C_0 \rrbracket_u(\llbracket C_1 \rrbracket_{uk})(s) \\ \llbracket \text{diverge} \rrbracket_{uks} &= \text{undefined} \\ \text{cond}_{\text{compl}}(b, k_0, k_1)(s) &= \begin{cases} k_0(s), & \text{if } b(s) = \text{true} \\ k_1(s), & \text{if } b(s) = \text{false} \\ \text{undefined}, & \text{if } b(s) = \perp \end{cases} \\ \text{cond}_{\text{comm}}(b, c_0, c_1)(k) &= \text{cond}_{\text{compl}}(b, c_0(k), c_1(k)) \end{aligned}$$

**Table 12.** Continuation semantics

Intuitively, the equation for command sequencing states that, to execute  $C_0 ; C_1$  with continuation  $k$ , execute  $C_0$  with a continuation that executes

$C_1$  with continuation  $k$ . Note that the first two equations can be simplified to

$$\begin{aligned}\llbracket \text{skip} \rrbracket u &= \text{id}_{(S \rightarrow O)} \\ \llbracket C_0 ; C_1 \rrbracket u &= \llbracket C_0 \rrbracket u \cdot \llbracket C_1 \rrbracket u\end{aligned}$$

It can be verified by a structural induction that, for any command  $C$  that does not involve completions,  $\llbracket C \rrbracket uk$  is equal to  $\llbracket C \rrbracket u; k$ , where, in the latter,  $\llbracket \cdot \rrbracket$  is the ‘direct’ (i.e., non-continuation) semantic valuation for commands used previously. The coercion from completions to commands is then interpreted as follows:

$$\llbracket \text{compl} \vdash \text{comm} \rrbracket kk' = k$$

so that

$$\llbracket K \rrbracket_{\text{comm}}(u)(k')(s) = \llbracket K \rrbracket_{\text{compl}}(u)(s).$$

The program output will be produced by executing the completion and the ‘normal’ command continuation  $k'$  is simply ignored.

Complete programs are now interpreted as follows:

$$\llbracket C \rrbracket_{\text{prog}} = g_0 ; \llbracket C \rrbracket_{\text{comm}}(u_0)(k_0)$$

where  $g_0$ ,  $k_0$ , and  $u_0$  are the appropriate initialization, completion, and initial environment, respectively, for all programs. The ‘initial’ continuation,  $k_0$ , will be applied to the *final* state of program execution, unless there is a ‘jump’ (such as an **abort**) or execution does not terminate.

At present, the only completions explicitly available for use by a programmer are the ones denoted by **stop** and **abort**, which terminate program execution. We can provide for more localized jumps by introducing procedural constants

$$\text{label, escape: } (\text{compl} \rightarrow \text{comm}) \rightarrow \text{comm}$$

and defining them so that the quantified commands

$$\# \text{label } \iota. C$$

and

$$\# \text{escape } \iota. C$$

are executed by executing  $C$  in the environment such that execution of  $\iota$  results in an immediate jump to the *beginning* or *end*, respectively, of  $C$ ; for the former, execution continues with  $C$  itself, and for the latter, execution continues with the rest of the program. Notice that the bound identifier  $\iota$  can be used in an actual parameter to a non-local procedure, and so it is



possible to ‘jump’ out of the body of a procedure to the (beginning or end of) the calling context.

As an example, the following code does a linear search for  $x$ :  $\mathbf{exp}[\tau]$  in an ‘array’  $f: \mathbf{exp}[\mathbf{nat}] \rightarrow \mathbf{exp}[\tau]$  over the range of ‘subscripts’ from  $a$  to  $b$ , both of type  $\mathbf{exp}[\mathbf{nat}]$ :

```
# new[bool] present.
(# escape xFound.
  (# iterate(a, b) i.
    if f(i) = x then
      present := true ; xFound);
  present := false)
{present = (x ∈ f(a..b))}
```

Quantifier *iterate* is as defined at the end of Section 5.1. The completion *xFound* is used to terminate the search as soon as a component equal to  $x$  is found.

The semantic equations for the new quantifier constants are as follows:

$$\llbracket \mathbf{label} \rrbracket_{upk} = \bigsqcup_{i \in \omega} f^i(\perp), \quad \text{where } f(k') = p(k')(k)$$

$$\llbracket \mathbf{escape} \rrbracket_{upk} = p(k)(k)$$

where  $p$  ranges over  $\llbracket \mathbf{compl} \rrbracket \rightarrow \llbracket \mathbf{comm} \rrbracket$ . From these we derive

$$\begin{aligned} \llbracket \# \mathbf{label} \iota. C \rrbracket_{uk} &= \llbracket \mathbf{label}(\lambda \iota: \mathbf{compl}. C) \rrbracket_{uk} \\ &= \llbracket \mathbf{label} \rrbracket_u(\llbracket \lambda \iota: \mathbf{compl}. C \rrbracket_u)k \\ &= \bigsqcup_{i \in \omega} f^i(\perp), \quad \text{where } f(k') = \llbracket C \rrbracket(u \mid \iota \mapsto k')(k), \end{aligned}$$

and

$$\begin{aligned} \llbracket \# \mathbf{escape} \iota. C \rrbracket_{uk} &= \llbracket \mathbf{escape}(\lambda \iota: \mathbf{compl}. C) \rrbracket_{uk} \\ &= \llbracket \mathbf{escape} \rrbracket_u(\llbracket \lambda \iota: \mathbf{compl}. C \rrbracket_u)(k) \\ &= \llbracket C \rrbracket(u \mid \iota \mapsto k)(k). \end{aligned}$$

Also, we can generalize the sequencing operator to completions, as follows:

*Sequencing:*

$$\frac{C: \mathbf{comm} \quad K: \mathbf{compl}}{C ; K: \mathbf{compl}}$$

with interpretation

$$\llbracket C ; K \rrbracket_{\mathbf{compl}}(u)(s) = \llbracket C \rrbracket_u(\llbracket K \rrbracket_u)s$$

**Exercise 6.6.1.** Verify that, although  $C ; K$  and **if**  $B$  **then**  $K_0$  **else**  $K_1$  (for completions  $K$ ,  $K_0$  and  $K_1$ ) can be syntactically analysed as *commands* in two ways, the semantic interpretations are not ambiguous.

## 6.6.2 Programming logic

We now consider the effect on our programming logic of introducing jumps into the programming language. First, we must re-interpret the Hoare-triple formula in this context. This is most conveniently done by introducing a new atomic formula, as follows:

*Hoare double:*

$$\frac{P: \text{assert} \quad K: \text{compl}}{\{P\}K: \text{spec}}$$

The ‘Hoare-double’ formula,  $\{P\}K$ , asserts that  $P$  is a sufficient pre-condition on initial states to ensure that terminating executions of  $K$  lead to acceptable program output (where, for example, an error message is *not* acceptable). We can then *define* the conventional Hoare triple for commands  $C$  by the following equivalence:

$$\{P\}C\{Q\} \equiv_{\text{spec}} \forall \iota: \text{compl.} \{Q\}\iota \Rightarrow \{P\}(C; \iota)$$

for  $\iota$  not free in  $P$ ,  $Q$  or  $C$ . In this context,  $\{P\}C\{Q\}$  asserts that  $P$  is sufficient to ensure that terminating executions of  $C$  lead to acceptable program output when supplied with a continuation for which  $Q$  is sufficient to ensure acceptable output on termination.

To define the semantics of the Hoare-double formula, let  $\phi$  be the characteristic function of the  $A$ -subset of  $O$ ; i.e.,  $\phi(o) = \text{true}$  just if  $o \in A$ . In effect,  $\phi$  is the (implicit) post-condition for *all* completions. We now interpret Hoare-double specifications as follows:

$$\llbracket \{P\}K \rrbracket u = \text{for all } s \in S \text{ and } o \in O, \llbracket P \rrbracket us \text{ and } \llbracket K \rrbracket us = o \text{ imply } \phi(o),$$

which is analogous to the interpretation of Hoare triples in ‘direct’ (i.e., non-continuation) semantics.

The continuation-semantic interpretation of the Hoare triple is now derivable as follows:

$$\begin{aligned} & \llbracket \{P\}C\{Q\} \rrbracket u \\ &= \llbracket \forall \iota: \text{compl.} \{Q\}\iota \Rightarrow \{P\}(C; \iota) \rrbracket u \quad (\iota \text{ not free in } P, Q, \text{ or } C) \\ &= \text{for all } k \in \llbracket \text{compl} \rrbracket, \\ & \quad \text{if} \quad \text{for all } s \in S \text{ and } o \in O, \\ & \quad \quad \llbracket Q \rrbracket us \text{ and } k(s) = o \text{ imply } \phi(o) \\ & \quad \text{then for all } s \in S \text{ and } o \in O, \\ & \quad \quad \llbracket P \rrbracket us \text{ and } \llbracket C \rrbracket uks = o \text{ imply } \phi(o). \end{aligned}$$

Surprisingly, the re-interpretation of the Hoare-triple formula does not necessitate substantial modification of the formal system. All of the axioms previously discussed remain valid in the present context, except for the following:

$$\begin{aligned} & \{P\}C\{Q_1\} \& \cdots \& \{P\}C\{Q_n\} \\ & \Rightarrow \{P\}C\{Q_1 \text{ and } \cdots \text{ and } Q_n\} \end{aligned}$$

For example, suppose  $C = \mathbf{abort}$  and  $n = 0$ ; the specification

$$\{P\}\mathbf{abort}\{\mathbf{true}\}$$

is *not* valid (for arbitrary  $P$ ) because the output will be *error*. The failure of this axiom seems to be the only objective evidence for the widely expressed opinion that reasoning about programs is more difficult if jumps are used. Another example of a formula that was valid in direct semantics, but is invalid in the present context, is the equivalence

$$C ; \mathbf{diverge} \equiv_{\text{comm}} \mathbf{diverge}$$

The Hoare-like axioms relevant to **stop**, **abort**, labels and escapes are as follows:

$$\{P\}\mathbf{stop}\{\mathbf{false}\}$$

$$\{\mathbf{false}\}\mathbf{abort}\{\mathbf{false}\}$$

$$(\forall \iota: \mathbf{compl.} \{P\}\iota\{\mathbf{false}\} \Rightarrow \{P\}C\{Q\}) \Rightarrow \{P\} \# \mathbf{label} \iota. C\{Q\}$$

$$(\forall \iota: \mathbf{compl.} \{Q\}\iota\{\mathbf{false}\} \Rightarrow \{P\}C\{Q\}) \Rightarrow \{P\} \# \mathbf{escape} \iota. C\{Q\}$$

where, in the latter two axioms,  $\iota$  should not be free in  $P$  or  $Q$ ; see [Reynolds, 1981a] for an example of a verification using an axiom similar to these. We can also state command equivalences such as

$$\# \mathbf{label} \iota. \iota \equiv \mathbf{diverge}$$

$$\# \mathbf{escape} \iota. \iota \equiv \mathbf{skip}$$

$$K ; C \equiv K$$

where  $C$  is a command and  $K$  is a completion.

## 6.7 Intermediate output

We can use continuations and a recursively-defined domain to describe the semantics of language features that produce ‘intermediate’ output, that is to say, output that is produced by a program *before* it terminates. To

illustrate this in the simplest possible way, we add the following form of command:

*Intermediate Output:*

tick: comm

The intention is that executing **tick** should have no effect on the computational state, but should append another ‘tick’ to the program output, like a clock. To model this, we assume continuation semantics for commands, as in Section 6.6, and re-define the domain of outputs as a least solution of the following isomorphism:

$$O \cong A + (\{tick\} \times O_{\perp}) + \{error\}$$

The elements of the resulting domain are essentially the finite sequences of ‘ticks’, terminated by an answer, *error*, or  $\perp$ , and a limit point, which represents the *infinite* sequence of ticks, modelling the output of non-terminating but ‘ticking’ programs. The ordering on this domain is determined by having a finite sequence of ticks terminated by  $\perp$  approximate any sequence obtained from it by replacing the  $\perp$  by any output.

The semantic equation for **tick** is then as follows:

$$\llbracket \mathbf{tick} \rrbracket_{uks} = \begin{cases} (tick, \perp), & \text{if } k(s) \text{ is undefined} \\ (tick, k(s)), & \text{otherwise,} \end{cases}$$

where we have not bothered to ‘tag’ elements of the domain of outputs.

## 6.8 Block expressions

Most programming languages allow command-like phrases to be components of expression-like ones. For example, consider adding a family of quantifier constants

$$\mathbf{result}[\tau]: ((\mathbf{exp}[\tau] \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl}) \rightarrow \mathbf{exp}[\tau]$$

such that the value of  $\# \mathbf{result}[\tau].K$  is the value of the expression to which the procedure  $\iota$  is applied during the execution of  $K$ : **compl**. This kind of facility is sometimes termed a *block expression*, but in many programming languages is allowed only in expression-procedure definitions.

Block expressions introduce the possibility of side effects to *non-local* variables during execution of an *expression*. We do not want to allow such side effects because of their drastic effects on the programming logic. But then the semantical problem is to restrict execution of the body of a block expression so that it cannot change the values of non-local variables, even if non-local commands or procedures are invoked.

Another problem with block expressions in the presence of jumps is that a non-local completion might be invoked, so that ‘evaluation’ of an expression might even fail to produce a value. If this were to be allowed, it would be necessary to re-define the domain of expression meanings as follows:

$$\llbracket \mathbf{exp}[\tau] \rrbracket = G \rightarrow \llbracket \mathbf{compl} \rrbracket$$

where  $G = \llbracket \tau \rrbracket \rightarrow \llbracket \mathbf{compl} \rrbracket$  is the domain of *expression continuations*. An expression meaning is then a function transforming an expression continuation into a command continuation; for example,

$$\begin{aligned} \llbracket N_0 + N_1 \rrbracket ug &= \llbracket N_0 \rrbracket ug_1 \\ &\text{where } g_1(n_0) = \llbracket N_1 \rrbracket ug_2 \\ &\text{where } g_2(n_1) = g(n_0 + n_1) \end{aligned}$$

where  $n_0, n_1 \in \llbracket \mathbf{nat} \rrbracket$ , and  $g, g_1, g_2 \in G$ , and

$$\llbracket \mathbf{result}[\tau] \rrbracket upg = p(q)$$

for all  $p \in \llbracket (\mathbf{exp}[\tau] \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl} \rrbracket$  and  $q$  is the element of  $\llbracket \mathbf{exp}[\tau] \rightarrow \mathbf{compl} \rrbracket$  such that  $q(e) = e(g)$  for all  $e \in \llbracket \mathbf{exp}[\tau] \rrbracket$ , so that

$$\llbracket \# \mathbf{result}[\tau] \iota. K \rrbracket ug = \llbracket K \rrbracket (u \mid \iota \mapsto q),$$

for  $q$  defined similarly.

Notice, however, that the interpretation of  $N_0 + N_1$  specifies the order of evaluation of the sub-expressions, and even equivalences such as

$$N_0 + N_1 \equiv N_1 + N_0$$

would fail in this framework, even if there are no side effects, since the two sub-expressions might jump to different places. In order to preserve the familiar ‘algebraic’ properties of expressions, we must instead try to give an interpretation that precludes non-local jumps (other than error stops) as well as side effects. This will be discussed in Section 7.7.

## 6.9 Bibliographic notes

The treatment of coercions in Section 6.1 is based on [Reynolds, 1980; Reynolds, 1981b; Reynolds, 1985; Oles, 1982; Oles, 1987; Reynolds, 1991]. The treatment of dynamic storage in Section 6.2 is based on [Scott and Strachey, 1971; Scott, 1972a; Strachey, 1972]. Many authors have expressed the view that a more abstract approach to local variables would be desirable: [Scott, 1972a; Donahue, 1977; Reynolds, 1981b; Oles, 1982; Halpern *et al.*, 1984; Oles, 1985; Brookes, 1985;



Meyer and Sieber, 1988]. The use of procedural parameters and product types to construct ‘objects’ with hidden representations is a well-known technique; see, for example, the Appendix of [Reynolds, 1978].

Acceptors are proposed in [Reynolds, 1980; Reynolds, 1981b]. Phrase types of the form  $\text{acc}[\tau] \ \& \ \text{exp}[\tau']$  (Section 6.5) are special cases of *conjunctive* types as introduced in [Coppo and Dezani, 1978] in the context of type assignment for untyped languages, and applied to explicitly-typed languages in [Reynolds, 1987; Reynolds, 1988]. Further discussion of data-structuring facilities in Algol-like languages may be found in [Tennent, 1989].

The first important application of recursive domain definitions was to obtain a semantic model of the untyped lambda calculus; this is described in [Scott, 1970; Scott, 1972b; Wadsworth, 1976; Barendregt, (revised edition) 1984]. The general method of solving domain equations is discussed in [Wand, 1979; Lehmann and Smyth, 1981; Smyth and Plotkin, 1982; Plotkin, 1985] and in the chapter on domain theory in this volume.

The use of continuations to treat the semantics of jumps was introduced in [Strachey and Wadsworth, 1974]. The ‘Hoare-double’ form of specification formula is from [Tennent and Tobin, 1991]; for related treatments, see [Clint and Hoare, 1972; de Bruin, 1980; Reynolds, 1981a].

## 7 Possible worlds

In this section we discuss a semantic technique that will allow us to solve three difficult problems that arose in earlier sections:

- a good semantics for local-variable declarations;
- a semantics for a ‘non-interference’ formula that allows Hoare-like reasoning about assignment commands in a language with procedures; and
- a semantics that allows commands to be used in expressions without causing side effects or non-local jumps.

The method is known as *possible-world semantics*, and is well-known to logicians, who have used it to interpret modal and intuitionistic logics.

Possible-world semantics is appropriate when it is necessary to import non-locally defined meanings into a context where some local conditions are applicable to the semantic domains and valuations. For example, a local-variable declaration is most appropriately thought of as temporarily ‘expanding’ the set of states from some non-local set  $S$  to a new local set  $S \times V$ , where  $V$  is the set of values possible for the new variable. In a conventional interpretation, non-local procedures and command meanings are defined for the non-local set  $S$  of states, and are not usable with the expanded local set of states  $S \times V$ . Possible-world interpretations will allow us to deal coherently with such changes of context.

It is very convenient to use a category-theoretical formulation of possible-world semantics. It is assumed in this section that the reader is familiar with elementary concepts of category theory, such as products, functors, and natural transformations.

## 7.1 Functor–category semantics

Suppose that  $X \in [\theta]_\pi$  and that  $w$  ranges over a set of objects, termed *possible worlds*, that determine certain ‘local’ aspects of the interpretation; that is, the sets or domains of environments and meanings and the valuation functions are now all ‘parameterized’ by possible worlds  $w$ :

$$[\pi]w \xrightarrow{[X]_{\pi\theta}(w)} [\theta]w$$

But the domains and valuation functions for different possible worlds should not be arbitrarily different from one another. To arrive at an appropriate uniformity condition, suppose that  $x$  is another possible world and that notation  $f: w \rightarrow x$  denotes one way of ‘changing’ from  $w$  to  $x$ . It is reasonable to require that, for any possible world  $w$ , there is a ‘null’ change-of-possible-world  $\text{id}_w: w \rightarrow w$  and that, if  $f: w \rightarrow x$  and  $g: x \rightarrow y$  are changes-of-possible-world, there is a composite change  $f; g: w \rightarrow y$  such that composition is associative and  $\text{id}_w$  is the identity. In short, possible worlds and changes-of-possible-world must form a *category*  $\mathbf{W}$ .

Now, a  $\mathbf{W}$ -morphism  $f: w \rightarrow x$  induces a change-of-meaning

$$[\theta]f: [\theta]w \rightarrow [\theta]x$$

for every phrase type  $\theta$ . Similarly,

$$[\pi]f: [\pi]w \rightarrow [\pi]x$$

should do the same kind of thing component-wise to  $\pi$ -compatible environments. It is reasonable to require that these induced mappings should preserve identities and composites. In short,  $[\pi]$  and  $[\theta]$  must be *functors* from  $\mathbf{W}$ , a category of possible worlds, to a suitable category of semantic domains:

$$\begin{array}{ccc} w & & [\pi]w \\ \downarrow f & & \downarrow [\pi]f \\ x & & [\pi]x \end{array} \quad \begin{array}{ccc} & & [\theta]w \\ & & \downarrow [\theta]f \\ & & [\theta]x \end{array}$$

Finally, the appropriate uniformity condition on the valuations is that, for every  $X \in [\theta]_\pi$ ,  $[X]_{\pi\theta}$  should be a *natural transformation* from  $[\pi]$  to  $[\theta]$ ; that is, the following diagram should commute for every  $\mathbf{W}$ -morphism  $f: w \rightarrow x$ :

$$\begin{array}{ccc}
 w & & \llbracket \pi \rrbracket w \xrightarrow{\llbracket X \rrbracket_{\pi\theta}(w)} \llbracket \theta \rrbracket w \\
 \downarrow f & & \downarrow \llbracket \pi \rrbracket f \quad \quad \downarrow \llbracket \theta \rrbracket f \\
 x & & \llbracket \pi \rrbracket x \xrightarrow{\llbracket X \rrbracket_{\pi\theta}(x)} \llbracket \theta \rrbracket x
 \end{array}$$

Note that this picture reduces to the conventional one when  $\mathbf{W}$  is the trivial (one-object and one-morphism) category.

The basic idea of the possible-world approach then is to move from the usual categories  $\mathbf{S}$ , of sets and functions, or  $\mathbf{D}$ , of domains and continuous functions, to more general *functor categories*  $\mathbf{W} \Rightarrow \mathbf{S}$  and  $\mathbf{W} \Rightarrow \mathbf{D}$  whose objects are the functors from a suitable (small) category  $\mathbf{W}$  of possible worlds to  $\mathbf{S}$  or to  $\mathbf{D}$ , and whose morphisms are the natural transformations of these functors.

## 7.2 Semantic-domain functors

In this section, we first define constructions on functors that are generalizations of the set and domain constructions used previously, and then show how to define semantic-domain functors  $\llbracket \tau \rrbracket$ ,  $\llbracket \theta \rrbracket$ , and  $\llbracket \pi \rrbracket$  for every data type  $\tau$ , phrase type  $\theta$ , and phrase-type assignment  $\pi$ .

If  $\mathbf{W}$  is *any* category (of possible worlds), we can define the following constructions of functors from  $\mathbf{W}$  to  $\mathbf{D}$  from functors  $D$  and  $E$  from  $\mathbf{W}$  to  $\mathbf{D}$ ;  $w$ ,  $x$ , and  $y$  are  $\mathbf{W}$ -objects, and  $f: w \rightarrow x$  and  $g: x \rightarrow y$  are  $\mathbf{W}$ -morphisms.

- $D \times E$ :

$$(D \times E)(w) = D(w) \times E(w),$$

and

$$(D \times E)(f)(a, b) = (D(f)(a), E(f)(b));$$

- $D_{\perp}$ :

$$D_{\perp}(w) = (D(w))_{\perp},$$

and

$$D_{\perp}(f)(a) = \begin{cases} \perp, & \text{if } a = \perp \\ D(f)(a), & \text{otherwise;} \end{cases}$$

- $D \rightarrow E$ :

$$(D \rightarrow E)(w)$$

$$= \left\{ m \in \prod_{f: w \rightarrow x} (D(x) \rightarrow E(x)) \mid \begin{array}{l} \text{for all } f: w \rightarrow x \text{ and } g: x \rightarrow y, \\ m(f); E(g) = D(g); m(f; g) \end{array} \right\},$$

ordered pointwise (i.e.,  $m_1 \sqsubseteq m_2$  iff  $m_1(f) \sqsubseteq m_2(f)$  for all  $f: w \rightarrow x$ ), where

$$\prod_{f: w \rightarrow x} \dots x \dots$$

here and in subsequent definitions is an abuse of notation for

$$\prod_{f \in \mathbf{W}(w, \cdot)} \dots \text{codom } f \dots,$$

where  $\text{codom } f$  is the co-domain of  $f$ ,  $\mathbf{W}(w, \cdot)$  is the set of all  $\mathbf{W}$ -morphisms with domain  $w$ , and

$$(D \rightarrow E)(f)(m)(g) = m(f; g);$$

To motivate the  $\rightarrow$  construction, consider that a procedure defined in possible world  $w$  might be called in any possible world  $x$  accessible from  $w$  using a  $\mathbf{W}$ -morphism  $f: w \rightarrow x$ , and it is the domain structure determined by  $x$  which should be in effect when the procedure body is executed. This suggests that we cannot just define  $(D \rightarrow E)(w)$  to be  $D(w) \rightarrow E(w)$ ; the meaning of a procedure defined in possible world  $w$  must be a *family* of functions, indexed by  $\mathbf{W}$ -morphisms  $f: w \rightarrow x$ . But such families of functions must be appropriately *uniform*; the uniformity condition is commutativity of all diagrams

$$\begin{array}{ccc} D(x) & \xrightarrow{m(f: w \rightarrow x)} & E(x) \\ \downarrow D(g: x \rightarrow y) & & \downarrow E(g: x \rightarrow y) \\ D(y) & \xrightarrow{m(f; g: w \rightarrow y)} & E(y) \end{array}$$

For  $D \rightarrow E$ , the construction is similar, but the uniformity condition only requires commutativity when the result of the partial mapping along the top of the diagram is defined:

$$\begin{aligned} & (D \rightarrow E)(w) \\ &= \left\{ m \in \prod_{f: w \rightarrow x} (D(x) \rightarrow E(x)) \mid \begin{array}{l} \text{for all } f: w \rightarrow x \text{ and } g: x \rightarrow y, \\ m(f); E(g) \subseteq D(g); m(f; g) \end{array} \right\}, \end{aligned}$$

ordered pointwise, where the  $\subseteq$  relation on partial functions is graph inclusion, and

$$(D \multimap E)(f)(m)(g) = m(f; g).$$

Intuitively, a function defined at some argument in one world should also be defined and have a corresponding result for a corresponding argument at all derived worlds.

Finally, if  $I$  is a finite set and, for every  $i \in I$ ,  $D_i$  is a functor from  $\mathbf{W}$  to  $\mathbf{D}$ , then we define  $\prod_{i \in I} D_i$  by

$$\left( \prod_{i \in I} D_i \right)(w) = \prod_{i \in I} D_i(w),$$

and

$$\left( \prod_{i \in I} D_i \right)(f)(d)(j) = D_j(f)(d_j).$$

For functors to  $\mathbf{S}$ , rather than  $\mathbf{D}$ , the constructions are defined in the same way, but are based on the corresponding construction in  $\mathbf{S}$ , and the component sets are not partially ordered. It is easily verified that these are all functors, and that the constructions reduce to the conventional ones when  $\mathbf{W}$  is trivial. It can be shown that, for *any*  $\mathbf{W}$ , functor categories  $\mathbf{W} \Rightarrow \mathbf{S}$  and  $\mathbf{W} \Rightarrow \mathbf{D}$  are Cartesian closed, and that the  $\times$  and  $\multimap$  defined above construct product and exponential objects, respectively, in these categories.

We will also use these constructions on *contravariant* functors. The product and lifting operations construct contravariant functors when applied to contravariant functors. For the exponentiation operations, the uniformity conditions are obtained by reversing the appropriate arrows and, for  $\multimap$ , reversing the partial ordering if the 'argument' functor is contravariant; for example, for  $D$  and  $E$  contravariant,

$$(D \multimap E)(w) = \left\{ m \in \prod_{f: w \rightarrow x} (D(x) \multimap E(x)) \mid \begin{array}{l} \text{for all } f: w \rightarrow x \text{ and } g: x \rightarrow y, \\ m(f; g); E(g) \subseteq D(g); m(f) \end{array} \right\},$$

so that commutativity of

$$\begin{array}{ccc} D(x) & \xrightarrow{m(f: w \rightarrow x)} & E(x) \\ \uparrow D(g: x \rightarrow y) & & \uparrow E(g: x \rightarrow y) \\ D(y) & \xrightarrow{m(f; g: w \rightarrow y)} & E(y) \end{array}$$

is required whenever the partial function at the *bottom* gives a defined result. The morphism parts are defined as for covariant functors, so that



$D \rightarrow E$  and  $D \multimap E$  are always covariant, even when  $D$  or  $E$  is contravariant.

We now show how, for any category  $\mathbf{W}$  of possible worlds, we can define functors  $\llbracket \tau \rrbracket$ ,  $\llbracket \theta \rrbracket$ , and  $\llbracket \pi \rrbracket$  for every data type  $\tau$ , phrase type  $\theta$ , and phrase-type assignment  $\pi$ . We can actually use exactly the same definitions as before, but with the names of primitive sets and domains re-interpreted as being primitive functors, and the set and domain constructions re-interpreted as the functor constructions defined above.

For every data type  $\tau$ ,  $\llbracket \tau \rrbracket$  is a constant functor (covariant or contravariant, as appropriate) such that, for every  $\mathbf{W}$ -object  $w$ ,  $\llbracket \tau \rrbracket w$  is the set of values of type  $\tau$ . Then, if  $S$  is a (possibly contravariant) functor such that  $S(w)$  is the set of computational states appropriate to possible world  $w$ , the definitions of Table 13, which in earlier chapters were definitions of *sets* or *domains*, may now be re-interpreted as definitions of direct-semantic *functors* from  $\mathbf{W}$  to  $\mathbf{S}$ , or to  $\mathbf{D}$ , as appropriate.

$$\begin{aligned}
 \llbracket \text{val}[\tau] \rrbracket &= \llbracket \tau \rrbracket_{\perp} \\
 \llbracket \text{exp}[\tau] \rrbracket &= S \multimap \llbracket \text{val}[\tau] \rrbracket \\
 \llbracket \text{assert} \rrbracket &= S \multimap \llbracket \text{bool} \rrbracket \\
 \llbracket \text{comm} \rrbracket &= S \multimap S \\
 \llbracket \text{acc}[\tau] \rrbracket &= \llbracket \text{exp}[\tau] \rrbracket \rightarrow \llbracket \text{comm} \rrbracket \\
 \llbracket \text{var}[\tau] \rrbracket &= \llbracket \text{acc}[\tau] \rrbracket \times \llbracket \text{exp}[\tau] \rrbracket \\
 \llbracket \theta \rightarrow \theta' \rrbracket &= \llbracket \theta \rrbracket \rightarrow \llbracket \theta' \rrbracket \\
 \llbracket \iota_1 \mapsto \theta_1 \ \&\ \dots \ \&\ \iota_n \mapsto \theta_n \rrbracket &= \prod_{1 \leq i \leq n} \llbracket \theta_i \rrbracket \\
 \llbracket \pi \rrbracket &= \prod_{\iota \in \text{dom } \pi} \llbracket \pi(\iota) \rrbracket
 \end{aligned}$$

**Table 13.** Semantic-domain functor

Note that expression, assertion, acceptor, and command meanings are, like procedures, *families* of functions, indexed by changes of possible worlds.

Similarly, if  $O$  is re-defined to be a suitable output functor, the following functor definitions are appropriate for continuation semantics:

$$\begin{aligned}
 \llbracket \text{compl} \rrbracket &= S \multimap O \\
 \llbracket \text{comm} \rrbracket &= \llbracket \text{compl} \rrbracket \rightarrow \llbracket \text{compl} \rrbracket
 \end{aligned}$$

The functor  $\llbracket \text{spec} \rrbracket$  for specifications will be discussed in Section 7.5.

### 7.3 Semantic valuations

It remains to define a *natural transformation*

$$\llbracket X \rrbracket_{\pi\theta} : \llbracket \pi \rrbracket \multimap \llbracket \theta \rrbracket$$

for every phrase  $X \in [\theta]_{\pi}$ .

$$\begin{aligned}
\llbracket \text{if } K \text{ then } X_0 \text{ else } X_1 \rrbracket wu &= \begin{cases} \llbracket X_0 \rrbracket wu, & \text{if } \llbracket K \rrbracket wu = \text{true} \\ \llbracket X_1 \rrbracket wu, & \text{if } \llbracket K \rrbracket wu = \text{false} \\ \perp, & \text{if } \llbracket K \rrbracket wu = \perp \end{cases} \\
\llbracket \text{if } B \text{ then } X_0 \text{ else } X_1 \rrbracket_\theta(w)(u) &= \text{cond}_\theta(w)(\llbracket B \rrbracket wu, \llbracket X_0 \rrbracket_\theta(w)(u), \llbracket X_1 \rrbracket_\theta(w)(u)) \\
\llbracket \text{true} \rrbracket wu &= \text{true} \\
\llbracket \text{not } B \rrbracket wufs &= \begin{cases} \text{false}, & \text{if } \llbracket B \rrbracket wufs = \text{true} \\ \text{true}, & \text{if } \llbracket B \rrbracket wufs = \text{false} \\ \perp, & \text{if } \llbracket B \rrbracket wufs = \perp \end{cases} \\
\llbracket \text{skip} \rrbracket wuf &= \text{id}_{S(x)} \\
\llbracket C_0 ; C_1 \rrbracket wuf &= \llbracket C_0 \rrbracket wuf ; \llbracket C_1 \rrbracket wuf \\
\llbracket A := E \rrbracket wu &= \llbracket A \rrbracket wu(\text{id}_w)(\llbracket E \rrbracket wu) \\
\llbracket \iota \rrbracket wu &= u(\iota) \\
\llbracket \lambda \iota. \theta. P \rrbracket wufa &= \llbracket P \rrbracket x(\llbracket \pi \rrbracket fu \mid \iota \mapsto a), \text{ for } a \in \llbracket \theta \rrbracket x \\
\llbracket P Q \rrbracket wu &= \llbracket P \rrbracket wu(\text{id}_w)(\llbracket Q \rrbracket wu) \\
\llbracket \text{forall}[\tau] \rrbracket wufpgs &= \text{for all } v \in \llbracket \text{val}[\tau] \rrbracket x, p(\text{id}_x)(v)(g)(s), \\
&\quad \text{for } p \in \llbracket \text{val}[\tau] \rightarrow \text{assert} \rrbracket x, g: x \rightarrow y, s \in S(y) \\
\llbracket \text{rec}[\theta] \rrbracket wufpg &= \bigsqcup_{i \in \omega} (p(g))^i(\perp_\theta), \text{ for } p \in \llbracket \theta \rightarrow \theta \rrbracket x, g: x \rightarrow y
\end{aligned}$$

Table 14. Semantic equations

In this section we discuss the valuations for several of the constructs discussed earlier. These do not depend on the category  $\mathbf{W}$  of possible worlds. The semantic equations for direct semantics are given in Table 14;  $w$  and  $x$  are  $\mathbf{W}$ -objects,  $u \in \llbracket \pi \rrbracket w$ ,  $f: w \rightarrow x$  is a  $\mathbf{W}$ -morphism,  $s \in S(x)$ ,  $K \in [\text{val}[\text{bool}]]_\pi$ ,  $B \in [\text{exp}[\text{bool}]]_\pi$ , and natural transformations

$$\text{cond}_\theta: [\text{exp}[\text{bool}]] \times [\theta] \times [\theta] \rightarrow [\theta]$$

may be defined by induction on  $\theta$  as follows:

$$\begin{aligned}
\text{cond}_{\text{exp}[\tau]}(w)(b, e_0, e_1)(f)(s) &= \begin{cases} e_0(f)(s), & \text{if } b(f)(s) = \text{true} \\ e_1(f)(s), & \text{if } b(f)(s) = \text{false} \\ \perp, & \text{if } b(f)(s) = \perp \end{cases} \\
\text{cond}_{\text{assert}}(w)(b, p_0, p_1)(f)(s) &= \begin{cases} p_0(f)(s), & \text{if } b(f)(s) = \text{true} \\ p_1(f)(s), & \text{if } b(f)(s) = \text{false} \\ \text{false}, & \text{if } b(f)(s) = \perp \end{cases} \\
\text{cond}_{\text{comm}}(w)(b, c_0, c_1)(f)(s) &= \begin{cases} c_0(f)(s), & \text{if } b(f)(s) = \text{true} \\ c_1(f)(s), & \text{if } b(f)(s) = \text{false} \\ \text{undefined}, & \text{if } b(f)(s) = \perp \end{cases} \\
\text{cond}_{\text{acc}[\tau]}(w)(b, a_0, a_1)(f)(e) &= \text{cond}_{\text{comm}}(x)(\llbracket \text{exp}[\text{bool}] \rrbracket fb, a_0(f)(e), a_1(f)(e))
\end{aligned}$$

$$\begin{aligned}
& \text{cond}_{\mathbf{var}[\tau]}(w)(b, (a_0, e_0), (a_1, e_1)) = (a, e) \\
& \text{where } a = \text{cond}_{\mathbf{acc}[\tau]}(w)(b, a_0, a_1) \\
& \text{and } e = \text{cond}_{\mathbf{exp}[\tau]}(w)(b, e_0, e_1) \\
& \text{cond}_{\theta \rightarrow \theta'}(w)(b, p_0, p_1)(f)(a) \\
& = \text{cond}_{\theta'}(x) \left( \llbracket \mathbf{exp}[\mathbf{bool}] \rrbracket f b, p_0(f)(a), p_1(f)(a) \right), \text{ for } a \in \llbracket \theta \rrbracket x.
\end{aligned}$$

It can be verified by a simultaneous structural induction that all of these valuations define natural transformations, and that all of the uniformity conditions are satisfied. For example, to show that  $\llbracket P Q \rrbracket$  is natural, where  $P \in [\theta \rightarrow \theta']_\pi$  and  $Q \in [\theta]_\pi$ , consider any  $\mathbf{W}$ -morphism  $f: w \rightarrow x$ ; we must show that the following diagram commutes:

$$\begin{array}{ccc}
\llbracket \pi \rrbracket w & \xrightarrow{\llbracket P Q \rrbracket_{\pi \theta'}(w)} & \llbracket \theta' \rrbracket w \\
\llbracket \pi \rrbracket f \downarrow & & \downarrow \llbracket \theta' \rrbracket f \\
\llbracket \pi \rrbracket x & \xrightarrow{\llbracket P Q \rrbracket_{\pi \theta'}(x)} & \llbracket \theta' \rrbracket x
\end{array}$$

Consider any  $u \in \llbracket \pi \rrbracket$ ; then

$$\begin{aligned}
& \llbracket P Q \rrbracket x (\llbracket \pi \rrbracket f u) \\
& = \llbracket P \rrbracket x (\llbracket \pi \rrbracket f u) (\text{id}_x) \left( \llbracket Q \rrbracket x (\llbracket \pi \rrbracket f u) \right) \\
& = \llbracket \theta \rightarrow \theta' \rrbracket f (\llbracket P \rrbracket w u) (\text{id}_x) \left( \llbracket \theta \rrbracket f (\llbracket Q \rrbracket w u) \right) \\
& \quad (\text{by induction}) \\
& = \llbracket P \rrbracket w u f \left( \llbracket \theta \rrbracket f (\llbracket Q \rrbracket w u) \right) \\
& \quad (\text{by the definition of } \llbracket \theta \rightarrow \theta' \rrbracket f) \\
& = \llbracket \theta' \rrbracket f (\llbracket P \rrbracket w u (\text{id}_w)) (\llbracket Q \rrbracket w u) \\
& \quad (\text{by the definition of } \llbracket \theta \rightarrow \theta' \rrbracket w) \\
& = \llbracket \theta' \rrbracket f (\llbracket P Q \rrbracket w u),
\end{aligned}$$

where the second-last step is justified by the commutativity of

$$\begin{array}{ccc}
\llbracket \theta \rrbracket w & \xrightarrow{\llbracket P \rrbracket w u (\text{id}_w)} & \llbracket \theta' \rrbracket w \\
\llbracket \theta \rrbracket f \downarrow & & \downarrow \llbracket \theta' \rrbracket f \\
\llbracket \theta \rrbracket x & \xrightarrow{\llbracket P \rrbracket w u f} & \llbracket \theta' \rrbracket x
\end{array}$$

To show that  $\llbracket \lambda \iota: \theta. P \rrbracket w u$  is a uniform family of functions, where  $P \in [\theta']_{(\pi|_{\iota \rightarrow \theta})}$  and  $u \in \llbracket \pi \rrbracket w$ , consider any  $\mathbf{W}$ -morphisms  $f: w \rightarrow x$  and

$g: x \rightarrow y$ ; we must verify the commutativity of the following diagram:

$$\begin{array}{ccc}
 \llbracket \theta \rrbracket x & \xrightarrow{\llbracket \lambda \iota: \theta. P \rrbracket wu f} & \llbracket \theta' \rrbracket x \\
 \llbracket \theta \rrbracket g \downarrow & & \downarrow \llbracket \theta' \rrbracket g \\
 \llbracket \theta \rrbracket y & \xrightarrow{\llbracket \lambda \iota: \theta. P \rrbracket wu(f; g)} & \llbracket \theta' \rrbracket y
 \end{array}$$

Consider any  $a \in \llbracket \theta \rrbracket x$ ; then

$$\begin{aligned}
 & \llbracket \lambda \iota: \theta. P \rrbracket wu(f; g)(\llbracket \theta \rrbracket ga) \\
 &= \llbracket P \rrbracket y(\llbracket \pi \rrbracket(f; g)u \mid \iota \mapsto \llbracket \theta \rrbracket ga) \\
 &= \llbracket \theta' \rrbracket g\left(\llbracket P \rrbracket x(\llbracket \pi \rrbracket fu \mid \iota \mapsto a)\right) \\
 &= \llbracket \theta' \rrbracket g(\llbracket \lambda \iota: \theta. P \rrbracket wu f a),
 \end{aligned}$$

where the second-last step is justified by the commutativity of the following diagram:

$$\begin{array}{ccc}
 \llbracket \pi' \rrbracket x & \xrightarrow{\llbracket P \rrbracket x} & \llbracket \theta' \rrbracket x \\
 \llbracket \pi' \rrbracket g \downarrow & & \downarrow \llbracket \theta' \rrbracket g \\
 \llbracket \pi' \rrbracket y & \xrightarrow{\llbracket P \rrbracket y} & \llbracket \theta' \rrbracket y
 \end{array}$$

where  $\pi' = (\pi \mid \iota \mapsto \theta)$ , because  $\llbracket P \rrbracket$  is a natural transformation.

It can also be verified that the Coherence and Substitution lemmas hold. Continuation-semantic valuations for the commands and completions would be similar.

## 7.4 Semantics of local variables

In this section, we begin to exploit the additional flexibility available to us in the possible-world framework by treating local-variable declarations. Intuitively, the effect of a local-variable declaration is to expand the set of states to allow for use of the new variable during an execution of the block body. Thus, if  $S$  is the set of states for an execution of a block  $\# \mathbf{new}[\tau] \iota. C$ , the meaning of the block should be, as usual, a partial function on  $S$ , but this will be defined in terms of the meaning of  $C$  when the set of allowed states is  $S \times V_\tau$ , where  $V_\tau$  is the set of values of type  $\tau$ . Notice, however, that the storage structure relevant to a procedure is that of the *call*, rather than the *definition*, of the procedure, and similarly if a command meaning is defined in one context and used in another.

We can formalize these intuitions by using a possible-world semantics in which the possible worlds are sets of states, and the changes of possible world are 'expansions' from any set  $S$  to  $S \times V_\tau$ . We define a category **W** of possible worlds as follows. The objects are sets,  $W, X, \dots$ , interpreted as sets of states representable by the run-time stack; the morphisms from  $W$  to  $X$  are pairs  $f, Q$  where  $f$  is a function from  $X$  to  $W$  and  $Q$  is an equivalence relation on  $X$  such that the following is a product diagram in **S**:

$$W \xleftarrow{f} X \xrightarrow{q} X/Q$$

where  $X/Q$  is the set of  $Q$ -equivalence classes of  $X$  and  $q$  maps every element of  $X$  to its  $Q$ -equivalence class. Intuitively,  $f$  extracts the small stack embedded in a larger one, and  $Q$  relates large stacks with identical 'expansion components'. This is just the category-theoretic way of saying that larger stacks are formed from smaller ones by adding independent components for local variables. It can be verified that  $X \cong W \times X/Q$  and that  $f$  is bijective (a one-to-one correspondence) on  $Q$ -equivalence classes.

The identity morphism  $\text{id}_W$  on an object  $W$  has as its two components: the identity function on  $W$ , and the universally true binary relation on  $W$ . The composition (in diagrammatic order)  $(f, Q); (g, R): W \rightarrow Z$  of morphisms  $f, Q: W \rightarrow X$  and  $g, R: X \rightarrow Y$  has as its two components: the functional composition of  $f$  and  $g$ , and the equivalence relation on  $Y$  that relates  $s_0, s_1 \in Y$  just if they are  $R$ -related and  $Q$  relates  $g(s_0)$  and  $g(s_1)$ . To see that these components have the required product property, let  $h, S = (f, Q); (g, R)$  and note that

$$\begin{aligned} Y &\cong X \times Y/R \\ &\cong (W \times X/Q) \times Y/R \\ &\cong W \times (X/Q \times Y/R) \\ &\cong W \times Y/S, \end{aligned}$$

by the isomorphism between  $Y/S$  and  $X/Q \times Y/R$  established by the function mapping  $[y]_S$  (i.e., the  $S$ -equivalence class of  $y$ ) to  $([g(y)]_Q, [y]_R)$ , whose inverse is the function mapping  $([x]_Q, [y]_R)$  to  $\{y' \in [y]_R \mid g(y')Qx\}$ .

An important kind of morphism in this category is 'expansion by a set'. If  $V$  is a set, then the morphism  $\times V: W \rightarrow W \times V$  has as its components: the projection function from  $W \times V$  to  $W$ , and the equivalence relation that relates  $(s_0, v_0)$  and  $(s_1, v_1)$  just if  $v_0 = v_1$ . Strictly speaking, the notation  $\times V$  is inadequate because the domain and co-domain objects for the expansion morphism are not uniquely determined; but these will always be evident from context. The state-set functor  $S$  for this category of possible worlds is defined by  $S(W) = W$ , discretely-ordered, and  $S(f, Q) = f$ . Notice that  $S$  is *contravariant*.



We now consider local-variable declarations. First, we define a functor  $expand_\tau: \mathbf{W} \rightarrow \mathbf{W}$  for each data type  $\tau$  as follows:

$$expand_\tau(W) = W \times \llbracket \tau \rrbracket W$$

and

$$expand_\tau(f, Q: W \rightarrow X) = f_\tau, Q_\tau$$

$$\text{where } f_\tau(x, v) = (f(x), v)$$

$$\text{and } (x_0, v_0)Q_\tau(x_1, v_1) \text{ if and only if } x_0Qx_1.$$

For any  $\mathbf{W}$ -morphism  $f, Q: W \rightarrow X$ , the following diagram commutes:

$$\begin{array}{ccc} W & \xrightarrow{\times \llbracket \tau \rrbracket W} & expand_\tau(W) \\ f, Q \downarrow & \sim & \downarrow expand_\tau(f, Q) \\ X & \xrightarrow{\times \llbracket \tau \rrbracket X} & expand_\tau(X) \end{array}$$

We can now define the acceptor and expression components

$$a_\tau(W) \in \llbracket \mathbf{acc}[\tau] \rrbracket (expand_\tau(W))$$

and

$$e_\tau(W) \in \llbracket \mathbf{exp}[\tau] \rrbracket (expand_\tau(W))$$

of a 'new' local variable of data type  $\tau$  in an expanded possible world  $expand_\tau(W)$ ; we assume direct semantics. If  $f, Q: expand_\tau(W) \rightarrow X$  and  $g, R: X \rightarrow Y$  are  $\mathbf{W}$ -morphisms, and  $e \in \llbracket \mathbf{exp}[\tau] \rrbracket X$ ,

$$\begin{aligned} a_\tau(W)(f, Q)(e)(g, R)(y_0 \in Y) \\ = \begin{cases} \text{undefined,} & \text{if } e(g, R)(y_0) = \perp \\ y_1, & \text{if } e(g, R)(y_0) = v_1 \in \llbracket \tau \rrbracket X \end{cases} \end{aligned}$$

where

(i)  $y_0 R y_1$  and  $g(y_0) Q g(y_1)$ ; and

(ii)  $f(g(y_1)) = (w, v_1)$ , where  $(w, v_0) = f(g(y_0))$ ;

and

$$e_\tau(W)(f, Q)(x_0 \in X) = v, \text{ where } (w, v) = f(x_0).$$

In the definition of  $a_\tau$ , a state  $y_1$  satisfying the two conditions must exist and be unique by the 'product' property of  $\mathbf{W}$ -morphism  $(f, Q); (g, R)$ . Intuitively, the effect of assigning to the acceptor is to replace the old value  $v_0$  in the appropriate component of the stack by a new value  $v_1$ , without

The variable-declaration quantifiers

for every data type  $\tau$  may now be defined as follows. Let  $w, x$  and  $y$  be  $\mathbf{W}$ -objects,  $f: w \rightarrow x$  and  $g: x \rightarrow y$  be  $\mathbf{W}$ -morphisms,  $u$  be any environment,  $p \in \llbracket \mathbf{var}[\tau] \rightarrow \mathbf{comm} \rrbracket x$ , and  $s \in S(y)$ ; then

where, as in Section 2.4,  $v_\tau$  is an initial value for all variables of type  $\tau$ . Then, using the syntactic equivalence for quantification and the valuations for abstraction and application, the derived direct-semantic interpretation of a block command declaring a local  $\tau$ -variable is as follows:

It is now possible to validate the equivalence of

and *c*. Unfortunately, this interpretation is not yet *fully* abstract. For example,

where  $n$  is not free in  $P$ :  $\mathbf{comm} \rightarrow \mathbf{comm}$  might set  $n$  to any value; however, because  $P$  cannot *read* from  $n$  and  $n$  is discarded after execution of the body, the block should be equivalent to  $P(\mathbf{skip})$ ; but this equivalence fails in the interpretation presented here. A refinement of this model that validates all test equivalences proposed so far is described in [O’Hearn and Tennent, 1993].

## 7.5 Specifications

We now consider how to define the semantic-domain functor  $\llbracket \text{spec} \rrbracket$  and interpret the logical operators in the possible-world context. It might seem reasonable to make  $\llbracket \text{spec} \rrbracket$  a constant functor with  $\llbracket \text{spec} \rrbracket w = \{\text{true}, \text{false}\}$  for all  $w$ . This does not work, however, because the truth value of a formula can change as the result of a change of possible world, and this would violate the uniformity requirement for natural transformations.

A fairly general solution (i.e., one suitable for *any* category of possible worlds) with acceptable *logical* properties is to define  $\llbracket \text{spec} \rrbracket w$  to be the set of all families of  $\mathbf{W}$ -morphisms with domain  $w$  satisfying the constraint that, if  $f: w \rightarrow x$  is in such a family, then so is  $f; g$  for *every*  $g: x \rightarrow y$ ; such a composition-closed family of morphisms is termed a *sieve on*  $w$ . Note that on any  $\mathbf{W}$ -object  $w$  there are always at least two sieves: the empty set of morphisms, and the family of all  $\mathbf{W}$ -morphisms with domain  $w$ . Intuitively, if the meaning of a formula in possible world  $w$  is a sieve on  $w$  and  $f: w \rightarrow x$  is an element of that sieve, then  $f$  is a change of possible world sufficient to make the formula true (and all *further* changes must maintain the truth of the formula). In effect, instead of recording the truth value of a formula at  $w$  alone, an element of  $\llbracket \text{spec} \rrbracket w$  records its truth value for every way of changing from possible world  $w$  (including, of course, the identity change).

An elegant way of formulating this construction is to let  $\llbracket \text{spec} \rrbracket$  be  $1 \rightarrow 1$ , where  $1: \mathbf{W} \rightarrow \mathbf{S}$  is the (covariant) constant functor defined by  $1(w) = \{*\}$  (or any other singleton set) for every  $w$ . The identity function on  $\{*\}$  may be identified with *true*, and the undefined function on  $\{*\}$  with *false*. Each element of  $\llbracket \text{spec} \rrbracket w$  is a family  $m$  of partial functions on  $\{*\}$  indexed by the  $\mathbf{W}$ -morphisms from  $w$  and subject to the condition that, if  $m(f)(*)$  is defined, then so is  $m(f; g)(*)$ ; in effect,  $m$  is just the ‘characteristic function’ of a sieve on  $w$ .

The valuations for the purely logical forms of specification in this framework are given in Table 15.

$$\begin{aligned}
 \llbracket \text{absurd} \rrbracket w f &= \text{false} \\
 \llbracket S_1 \ \& \ S_2 \rrbracket w f &= \llbracket S_1 \rrbracket w f \text{ and } \llbracket S_2 \rrbracket w f \\
 \llbracket S_1 \Rightarrow S_2 \rrbracket w f &= \text{for all } g: x \rightarrow y, \\
 &\quad \text{if } \llbracket S_1 \rrbracket w f(g) \text{ then } \llbracket S_2 \rrbracket w f(g) \\
 \llbracket \forall \iota: \theta. S \rrbracket w f &= \text{for all } g: x \rightarrow y \text{ and } m \in \llbracket \theta \rrbracket y, \\
 &\quad \llbracket S \rrbracket y(\llbracket \pi \rrbracket(f; g)(u) \mid \iota \mapsto m)(\text{id}_y)
 \end{aligned}$$

Table 15. Semantics of specifications

Note the ‘implicit’ quantifications over changes of possible world in the valuations for implication and quantification. These are required in general to ensure the following ‘monotonicity’ property, which is required by the naturality condition on valuations: for any specification  $S$ ,  $\llbracket S \rrbracket w f$  implies

$\llbracket S \rrbracket wu(f; g)$  for every appropriate  $g$ . It can be verified that all of the usual (intuitionistic) rules of inference preserve validity if these valuations are used. As a pure logic, intuitionistic logic is weaker than classical logic; but the axioms of an intuitionistic *theory* can be stronger than would be possible in a corresponding classical theory. This additional generality will prove to be essential in our applications, and this explains why, in previous sections, we adopted intuitionistic, rather than classical, logic for logical connectives.

We conclude this section by presenting the valuations for the atomic formulas that we have used in our programming logics. For equivalence and approximation,

$$\begin{aligned} \llbracket Z_1 \equiv_{\theta} Z_2 \rrbracket wu f &= \text{for all } g: x \rightarrow y, \llbracket \theta \rrbracket(f; g)(\llbracket Z_1 \rrbracket wu) = \llbracket \theta \rrbracket(f; g)(\llbracket Z_2 \rrbracket wu) \\ \llbracket Z_1 \sqsubseteq_{\theta} Z_2 \rrbracket wu f &= \text{for all } g: x \rightarrow y, \llbracket \theta \rrbracket(f; g)(\llbracket Z_1 \rrbracket wu) \sqsubseteq \llbracket \theta \rrbracket(f; g)(\llbracket Z_2 \rrbracket wu) \end{aligned}$$

For Hoare triples (using direct semantics):

$$\begin{aligned} \llbracket \{P\}C\{Q\} \rrbracket wu f &= \text{for all } s_0, s_1 \in S(x), \\ &\llbracket P \rrbracket wu f s_0 \text{ and } \llbracket C \rrbracket wu f s_0 = s_1 \text{ imply } \llbracket Q \rrbracket wu f s_1 \end{aligned}$$

and for Hoare doubles (using continuation semantics):

$$\begin{aligned} \llbracket \{P\}K \rrbracket wu f &= \text{for all } s \in S(x) \text{ and } o \in O(x), \\ &\llbracket P \rrbracket wu f s \text{ and } \llbracket K \rrbracket wu f s = o \text{ imply } \phi(o) \end{aligned}$$

All of the axioms for our Algol-like language are valid with these interpretations. For example, we will show the validity of the Eta law. Suppose that  $F \in [\theta \rightarrow \theta']_{\pi}$  with  $\iota \notin \text{dom } \pi$  and consider any  $u \in \llbracket \pi \rrbracket$ ; we must show that  $\llbracket \lambda \iota: \theta. F(\iota) \equiv_{\theta \rightarrow \theta'} F \rrbracket wu(\text{id}_w)$ . Consider any  $\mathbf{W}$ -morphism  $f: w \rightarrow x$ ; we now must show that

$$\llbracket \theta \rightarrow \theta' \rrbracket(f)(\llbracket \lambda \iota: \theta. F(\iota) \rrbracket wu) = \llbracket \theta \rightarrow \theta' \rrbracket(f)(\llbracket F \rrbracket wu).$$

Consider any  $\mathbf{W}$ -morphism  $g: x \rightarrow y$  and  $a \in \llbracket \theta \rrbracket y$ ; then

$$\begin{aligned} &\llbracket \theta \rightarrow \theta' \rrbracket(f)(\llbracket \lambda \iota: \theta. F(\iota) \rrbracket wu)(g)(a) \\ &= \llbracket \lambda \iota: \theta. F(\iota) \rrbracket wu(f; g)(a) \\ &= \llbracket F(\iota) \rrbracket y u', \quad \text{where } u' = (\llbracket \pi \rrbracket(f; g)u \mid \iota \mapsto a) \\ &= \llbracket F \rrbracket y u'(\text{id}_y)(\llbracket \iota \rrbracket y u') \\ &= \llbracket F \rrbracket y(\llbracket \pi \rrbracket(f; g)u)(\text{id}_y)(a) \quad (\text{because } \iota \notin \text{dom } \pi) \\ &= \llbracket \theta \rightarrow \theta' \rrbracket(f; g)(\llbracket F \rrbracket wu)(\text{id}_y)(a) \quad (\text{by the naturality of } \llbracket F \rrbracket) \\ &= \llbracket \theta \rightarrow \theta' \rrbracket(f)(\llbracket F \rrbracket wu)(g)(a) \quad (\text{by the definition of } \llbracket \theta \rightarrow \theta' \rrbracket f). \end{aligned}$$

## 7.6 Non-interference specifications

It was pointed out in Section 5.4 that Hoare-like reasoning about assignment commands in the presence of procedures is quite problematical because of the possibilities of interference through non-local variables, aliasing, and bad variables. In this section, we describe one approach to dealing with these issues. The basic idea is to introduce new atomic specification formulas as follows:

*Good variable:*

$$\frac{V: \mathbf{var}[\tau]}{\mathbf{gv}_\tau(V): \mathbf{spec}}$$

*Non-interference:*

$$\frac{C: \theta \quad E: \theta'}{C \# E: \mathbf{spec}}$$

Informally,  $\mathbf{gv}_\tau(V)$  asserts that  $V$  is a good variable, and  $C \# E$  asserts that every way of using  $C$  preserves any value produced by using  $E$ . For example, if  $C$  is a command and  $E$  is an assertion, then every execution of  $C$  preserves the truth or falsity of  $E$ ; if  $C$  is an acceptor, then no assignment to  $C$  can interfere with  $E$ ; and if  $C$  is a procedure, then no call of  $C$  can interfere with  $E$  other than by using an argument that interferes with  $E$ .

Then, an axiom scheme for assignment commands in the presence of procedures is

$$\pi \vdash \mathbf{gv}_\tau(V) \ \& \ V \# Q \Rightarrow \{Q(E)\}V := E\{Q(V)\},$$

when  $V \in [\mathbf{var}[\tau]]_\pi$ ,  $E \in [\mathbf{exp}[\tau]]_\pi$ , and  $Q \in [\mathbf{exp}[\tau] \rightarrow \mathbf{assert}]_\pi$ . The consequent of the axiom is essentially similar to Hoare's axiom discussed in Section 2.6, but, in the more general context, the antecedents guard against possible bad variables, interference through non-locals, or aliasing.

Of course, the formal system must provide ways to discharge good-variable and non-interference assumptions. The following axiom allows a non-interference specification to be 'decomposed' into a conjunction of non-interference formulas for its free identifiers:

*Non-interference decomposition:*

$$\begin{aligned} \pi \vdash \iota \# \iota' \text{ for every } \iota \in \mathbf{free}(X) \text{ and } \iota' \in \mathbf{free}(X') \\ \Rightarrow X \# X' \end{aligned}$$

Intuitively, this axiom captures the fact that there are no *anonymous* 'channels of interference' in our language.



The following axiom allows good-variable and non-interference assumptions about locally declared variables to be discharged:

*Local-variable declaration:*

$$\begin{aligned} \pi \vdash (\forall \iota: \mathbf{var}[\tau]. \mathbf{gv}_\tau(\iota) \ \& \cdots \ \& \iota \# E_i \ \& \cdots \ \& C_j \ \# \iota \ \& \cdots \\ &\Rightarrow \{P\}C\{Q\}) \\ &\Rightarrow \{P\} \# \mathbf{new}[\tau] \iota. C\{Q\}, \end{aligned}$$

when  $\iota \notin \text{dom } \pi$ ,  $P, Q \in [\mathbf{assert}]_\pi$ ,  $C \in [\mathbf{comm}]_{(\pi|\iota \mapsto \mathbf{var}[\tau])}$ , and the  $E_i$  and  $C_j$  are arbitrary phrases in  $\pi$ . Essentially, this states that the identifiers declared by local-variable declarations denote variables that

- are good variables;
- do not interfere with non-local entities; and
- are not interfered with by non-local entities.

Before discussing further axioms for non-interference formulas, consider the following ‘weak’ interpretation of  $C \# P$  when  $C$  is a command and  $P$  is an assertion (in conventional direct semantics):

$$\begin{aligned} \llbracket C \# P \rrbracket u \\ = \text{ for all } s_0, s_1 \in S, \text{ if } \llbracket C \rrbracket u s_0 = s_1 \text{ then } \llbracket P \rrbracket u s_1 = \llbracket P \rrbracket u s_0; \end{aligned}$$

that is, every *complete* execution of  $C$  leaves the value of assertion  $P$  unchanged. This would be satisfactory for some applications, but a stronger and more useful interpretation of  $C \# P$  is possible: intuitively, the value of  $P$  is to remain constant *throughout* the (terminating) executions of  $C$ . To see the motivation for this, consider the following axiom:

*Constancy:*

$$\pi \vdash C \# R \ \& \ (\{R\} \Rightarrow \{P\}C\{Q\}) \Rightarrow \{P \text{ and } R\}C\{Q \text{ and } R\}$$

when  $C \in [\mathbf{comm}]_\pi$  and  $P, Q, R \in [\mathbf{assert}]_\pi$ . This asserts that, if no execution of  $C$  interferes with assertion  $R$  and  $R$  holds before an execution of  $C$ , then  $R$  will continue to hold throughout the execution, and so it is possible to treat  $R$  as a static assertion in partial-correctness reasoning about  $C$ .

For example, suppose that  $C$  is a binary-search algorithm and  $R$  asserts that the array being searched is sorted; it is obvious by inspection that  $C \# R$ , and so the axiom justifies ‘factoring’  $R$  out of the pre- and post-conditions for the algorithm and *also* assuming  $R$  whenever necessary in the verification of  $C$ . With the weaker interpretation of non-interference, only the following weaker form of Constancy would be valid:

$$C \# R \ \& \ \{P\}C\{Q\} \Rightarrow \{P \text{ and } R\}C\{Q \text{ and } R\},$$

and  $R$  could not be used as a ‘local’ mathematical fact in reasoning about  $C$ .

Similarly, the following axiom requires the stronger interpretation of non-interference:

*Non-interference composition:*

$$\pi \vdash C \# E \ \& \ (X \# E \Rightarrow \{P\}C\{Q\}) \Rightarrow \{P\}C\{Q\}$$

when  $C \in [\mathbf{comm}]_\pi$  and  $P, Q \in [\mathbf{assert}]_\pi$ . It asserts that, if  $C$  does not interfere with  $E$ , then it is possible to assume that no phrase interferes with  $E$  in partial-correctness reasoning about  $C$ .

A small example will demonstrate the use of some of the axioms for good-variable and non-interference specifications. We can use *Constancy* and *Non-interference decomposition* to derive

$$c \# n \Rightarrow \{n = 0\}c\{n = 0\}$$

for  $c: \mathbf{comm}$  and  $n: \mathbf{var}[\mathbf{nat}]$ , and then conventional Hoare-logic axioms to derive

$$c \# n \ \& \ \mathbf{gv}(n) \Rightarrow \{\mathbf{true}\}n := 0; c; \mathbf{if} \ n = 0 \ \mathbf{then} \ \mathbf{diverge}\{\mathbf{false}\}$$

Finally, the *Local-variable declaration* axiom can be used to show that the following block  $B$

$$\begin{aligned} &\# \mathbf{new}[\mathbf{nat}] \ n. n := 0; \\ &\quad c; \\ &\quad \mathbf{if} \ n = 0 \ \mathbf{then} \ \mathbf{diverge} \end{aligned}$$

satisfies the specification  $\{\mathbf{true}\}B\{\mathbf{false}\}$ , and so is equivalent to **diverge**, as expected.

We now turn to a detailed consideration of the interpretation of non-interference formulas. At first sight, the strong interpretation  $C \# P$  seems to require treating the semantics of commands in an ‘operational’ way; i.e., dealing explicitly with ‘intermediate’ states of command executions. But the desired strong interpretation of  $C \# P$  can be specified in a more elegant and satisfactory way by using possible-world semantics. The category of possible worlds  $\mathbf{W}$  is re-defined so that the objects and morphisms are sets of (allowed) states and pairs  $f, Q$ , as before, but we now weaken the condition on  $f$  and  $Q$  so that  $f$  is required only to be *injective* on  $Q$ -equivalence classes; i.e.,  $f, Q: W \rightarrow X$  is a  $\mathbf{W}$ -morphism if and only if

- $f$  is a function from  $X$  to  $W$ ;
- $Q$  is an equivalence relation on  $X$ ; and
- for all  $x, x' \in X$ ,  $xQx'$  and  $f(x) = f(x')$  imply  $x = x'$ .

The identity morphisms and composites are as before.

We can show that, for  $h, S = (f, Q); (g, R): W \rightarrow Y$ ,  $h$  is injective on  $S$ -equivalence classes as follows. Consider  $y, y' \in Y$  such that  $ySy'$  and  $h(y) = h(y')$ ; then  $yRy'$  and  $g(y)Qg(y')$  by the definition of  $S$ , and  $f(g(y)) = f(g(y'))$  by the definition of  $h$ . By the injectivity of  $f$  on  $Q$ -equivalence classes,  $g(y) = g(y')$ , and then, by the injectivity of  $g$  on  $R$ -equivalence classes,  $y = y'$ .

In this richer category, there is, as previously, an expansion morphism,  $\times V: W \rightarrow W \times V$ , for any data type  $V$ , and also

- a *state-set restriction* morphism,  $\lceil W': W \rightarrow W'$ , for any  $W' \subseteq W$ , where the components are: the insertion function from  $W'$  to  $W$ , and the universally true binary relation on  $W'$ ; and
- a *state-change restriction* morphism,  $\lceil Q: W \rightarrow W$ , for any equivalence relation  $Q$  on  $W$ , where the components are: the identity function on  $W$ , and  $Q$ .

Intuitively,  $\lceil W'$  changes worlds so that command executions must stay within  $W'$  (or fail to terminate), and  $\lceil Q$  imposes the constraint that a command execution can change the state from  $s$  to  $s'$  only if  $sQs'$ . State-change restrictions are enforced by defining  $\llbracket \text{comm} \rrbracket$  to be the following *sub-functor* of  $S \rightarrow S$ :

$\llbracket \text{comm} \rrbracket w$

$$= \left\{ c \in (S \rightarrow S)(w) \mid \begin{array}{l} \text{for all } f: w \rightarrow x \text{ and } s \in S(x), \\ S(\lceil X' \rceil; c(f, Q)) = c((f, Q); \lceil X' \rceil; S(\lceil X' \rceil)) \\ \text{where } X' = \{s' \in S(x) \mid s'Qs\} \end{array} \right\},$$

where  $S$  is, as before, the state-set functor. The condition ensures that, for every  $\mathbf{W}$ -morphism  $f, Q: w \rightarrow x$ , execution of  $c(f, Q)$  stays within a  $Q$ -equivalence class. For the morphism part,

$$\llbracket \text{comm} \rrbracket(f, Q)(c)(g, R) = c((f, Q); (g, R))$$

and it can be verified that this defines a functor from  $\mathbf{W}$  to  $\mathbf{D}$ .

With  $\llbracket \tau \rrbracket$  as before, the remaining definitions of Table 13 define the other semantic-domain functors appropriate to our new category of possible worlds. The semantic equations of Section 7.3 may also be adopted without modification. For variable declarations, the only change from Section 7.4 is that, in the definition of  $a_\tau(W)$ , an updated state  $y_1 \in Y$  satisfying the two conditions need not exist, and then

$$a_\tau(W)(f, Q: W \rightarrow X)(e)(g, R: X \rightarrow Y)(y_0 \in Y)$$

must be undefined; but if such a state exists, the injectivity requirement on morphisms ensures that it is unique.

We now interpret  $C \# P$  for  $C: \theta$  and  $P: \text{assert}$  using a state-change restriction morphism as follows: for  $f: w \rightarrow x$  in  $\mathbf{W}$ ,

$$\llbracket C \# P \rrbracket wuf = \left( \llbracket \theta \rrbracket f(\llbracket C \rrbracket wu) = \llbracket \theta \rrbracket (f; \lceil R \rceil (\llbracket C \rrbracket wu)) \right)$$

where, for  $s, s' \in S(x)$ ,  $sRs'$  if and only if  $\llbracket P \rrbracket wufs = \llbracket P \rrbracket wufs'$ . Intuitively, the equation asserts that  $C$  does not interfere with  $P$  just if restricting  $C$  so that the value of  $P$  is invariant does not make the execution less defined. A use of  $C$  that, at some point, attempts to change the value of  $P$  will be non-terminating if this state-change is not ‘allowed’ by equivalence relation  $R$ , and this difference between the restricted and the unrestricted use of  $C$  indicates potential interference. A natural generalization of this interpretation also allows treating  $C \# E$  when  $E$  is *any* (possibly higher-order) expression-like phrase; we refer the reader to [O’Hearn, 1990; O’Hearn and Tennent, 1991] for details.

We can now use this interpretation of  $C \# E$  to give the following definition of good-variable specifications:

$$\begin{aligned} \mathbf{gv}_\tau(V) \equiv \quad & \forall \iota_0: \mathbf{exp}[\tau]. \forall \iota_1: \mathbf{exp}[\tau] \rightarrow \mathbf{assert}. \\ & V \# \iota_1 \Rightarrow \{\iota_1(\iota_0)\}V := \iota_0\{\iota_1(V)\} \end{aligned}$$

for any  $\iota_0$  and  $\iota_1$  not free in  $V$ . Intuitively, this states that a good variable is one for which Hoare’s assignment axiom is always valid (provided that the variable does not interfere inappropriately with the assertion).

Using these interpretations, it is possible to validate all of the axioms we have discussed; see [O’Hearn, 1990; O’Hearn and Tennent, 1991]. A key point for the validation of Non-interference composition and Constancy is that the implicit quantification in the possible-world valuation for intuitionistic implication allows restriction to a subset of states for which an assertion or an expression has a constant value, and, in fact, it can be shown that only trivial models would be possible if the non-constructive *Reductio ad Absurdum* were added to the logical rules.

## 7.7 Semantics of block expressions

To interpret block expressions (Section 6.8), it again seems necessary to use the possible-world approach, because we need to impose local constraints that preclude side effects to non-local variables and non-local jumps.

We first enrich the category of possible worlds to allow for ‘local’ answer domains. Consider the category whose objects are domains, interpreted as domains of local answers, and whose morphisms are injective continuous functions, interpreted as mapping non-local answers into their local representatives. Then we take  $\mathbf{W}$ , our category of possible worlds, to be the

product of the state-set category used before and this answer-domain category; that is, the objects of  $\mathbf{W}$  are pairs  $W, D$ , where  $W$  is the local set of states and  $D$  is the domain of local answers, and the morphisms from  $W, D$  to  $X, E$  have the form  $(f, Q), i$  where  $f$  is a function from  $X$  to  $W$ ,  $Q$  is an equivalence relation on  $X$ ,  $i$  is an injection from  $D$  to  $E$ , and  $f$  is injective on  $Q$ -equivalence classes.

In this category, we can define, for every object  $W, D$ , a *state-set restriction* morphism

$$[W': W, D \rightarrow W', D$$

for any subset  $W'$  of  $W$ , a *state-change restriction* morphism

$$[Q: W, D \rightarrow W, D$$

for any equivalence relation  $Q$  on  $W$ , and a *state-set expansion* morphism

$$\times V: W, D \rightarrow (W \times V), D$$

for any set  $V$ , essentially as before, but ignoring the answer-related components. Furthermore, we can now define an *answer-domain adjunction* morphism

$$+D': W, D \rightarrow W, (D + D')$$

for any domain  $D'$  which is to be ‘added’ to the local-answer component;  $+D'$  is like the identity morphism except that the second component is the canonical injection from  $D$  into the domain sum  $D + D'$ .

We now re-define  $S$  to be the (contravariant) functor such that  $S(W, D) = W$  and  $S((f, Q), i) = f$ , and  $O$  to be the (covariant) functor such that  $O(W, D) = A + E + D$  and  $O((f, Q), i) = \text{id}_A + \text{id}_E + i$ , where  $A$  is the domain of *program answers* and  $E$  is the domain of ‘erroneous outputs’. Then, as before, we have  $\llbracket \text{compl} \rrbracket = S \multimap O$ , so that the uniformity condition on any  $k \in (S \multimap O)(w)$  requires commutativity of

$$\begin{array}{ccc} S(x) & \xrightarrow{k(f)} & O(x) \\ S(g) \uparrow & & \downarrow O(g) \\ S(y) & \xrightarrow{k(f; g)} & O(y) \end{array}$$

for all changes of possible world  $f: w \rightarrow x$  and  $g: x \rightarrow y$  whenever  $k(f; g)$  gives a defined result.

In this context,  $\phi$ , the ‘post-condition’ for outputs, must be the natural transformation from  $O$  to  $\llbracket \text{bool} \rrbracket$  whose components act as before on global outputs (elements of  $A$  or  $E$ ), and map all local answers to *true*. Then,

$$\llbracket \{P\}K \rrbracket wuf = \text{for all } s \in S(x) \text{ and } o \in O(x),$$



$\llbracket P \rrbracket wufs$  and  $\llbracket K \rrbracket wufs = o$  imply  $\phi_x(o)$

Finally, to treat the block-expression quantifier **result** $[\tau]$ , we define a functor  $adjoin_\tau: \mathbf{W} \rightarrow \mathbf{W}$  as follows:

$$adjoin_\tau(W, D) = W, (D + V)$$

$$adjoin_\tau((f, Q), i: W, D \rightarrow W', D') = (f, Q), i'$$

where  $i' = i + \text{id}_V$ , for  $V = \llbracket \tau \rrbracket(W, D)$ . For any change of possible worlds  $f: w \rightarrow x$ , the following diagram commutes:

$$\begin{array}{ccc} w & \xrightarrow{+\llbracket \tau \rrbracket w} & adjoin_\tau(w) \\ f \downarrow & & \downarrow adjoin_\tau(f) \\ x & \xrightarrow{+\llbracket \tau \rrbracket x} & adjoin_\tau(x) \end{array}$$

Then, if  $f: w \rightarrow x$  and  $g: x \rightarrow y$  are  $\mathbf{W}$ -morphisms,  $u$  is any environment, state  $s \in S(y)$ , and  $p \in \llbracket (\mathbf{exp}[\tau] \rightarrow \mathbf{compl}) \rightarrow \mathbf{compl} \rrbracket x$ ,

$$\begin{aligned} & \llbracket \mathbf{result}[\tau] \rrbracket wufpgs \\ &= \begin{cases} v, & \text{if } p(+\llbracket \tau \rrbracket x)(q)(adjoin_\tau(g; \lceil \{s\}))(s) = v \in \llbracket \tau \rrbracket x \\ \perp, & \text{otherwise,} \end{cases} \end{aligned}$$

where  $q \in \llbracket \mathbf{exp}[\tau] \rightarrow \mathbf{compl} \rrbracket x'$  for  $x' = adjoin_\tau(x)$  is defined as follows: for  $h: x' \rightarrow y$ ,  $e \in \llbracket \mathbf{exp}[\tau] \rrbracket y$ ,  $i: y \rightarrow w$ , and  $s' \in S(w)$ ,

$$q(h)(e)(i)(s') = \begin{cases} \text{undefined,} & \text{if } e(i)(s') = \perp; \\ O(h; i)(e(i)(s')) \text{ in } O(x'), & \text{otherwise.} \end{cases}$$

This yields the following interpretation for the block expression:

$$\begin{aligned} \llbracket \# \mathbf{result}[\tau] \iota. K \rrbracket wufs &= \begin{cases} v, & \text{if } \llbracket K \rrbracket w'u'f's = v \in \llbracket \tau \rrbracket x \text{ in } O(x'); \\ \perp, & \text{otherwise,} \end{cases} \\ & \text{where } w' = adjoin_\tau(w) \text{ and } x' = adjoin_\tau(x) \\ & \text{and } u' = \left( \llbracket \pi \rrbracket (+\llbracket \tau \rrbracket w)(u) \mid \iota \mapsto q \right) \\ & \text{and } f' = adjoin_\tau(f; \lceil \{s\} \rceil), \end{aligned}$$

and  $q \in \llbracket \mathbf{exp}[\tau] \rightarrow \mathbf{compl} \rrbracket w'$  is defined in the same way as the  $q$  above. The restriction to  $\{s\}$  precludes side effects, and the ability to discriminate between 'local' and 'non-local' answers allows us to produce  $\perp$  if there is a non-local jump.

It must be admitted, however, that the treatment of side effects given above is not entirely satisfactory. There are side effects that can be regarded as ‘benevolent’ in the sense that, when viewed from a sufficiently abstract point of view, they do *not* change the state. A typical example is ‘path compression’ while accessing the root of a tree when using the rooted-tree representation of a set partition; the change in the representation does not change the abstract partition being represented. Such benevolent side effects should be allowed, but it is not clear how this can be done without opening the door to non-benevolent ones as well.

It is also not clear how to reason about block expressions. The fundamental difficulty is that assertions about the value of a block expression must allow for the possibility that the body may fail to terminate (or attempt a side effect or non-local jump); but, in our intuitionistic partial-correctness framework, there is no way to express that completions or commands terminate. This is surprising, because  $\{\mathbf{true}\}C\{\mathbf{false}\}$  asserts that  $C$  never terminates. But  $\neg\{\mathbf{true}\}C\{\mathbf{false}\}$  (i.e.,  $\{\mathbf{true}\}C\{\mathbf{false}\} \Rightarrow \mathbf{absurd}$ ) asserts that, in every possible world,  $C$  sometimes terminates; this is false for *every*  $C$  because there are possible worlds with *empty* state-sets.

## 7.8 Bibliographic notes

Kripke’s semantics of intuitionistic logic is given in [Kripke, 1965]; see also [van Dalen, 1983; Dummett, 1977; Goldblatt, 1979]. Application of the functor-category approach to semantics of programming languages was initiated by [Reynolds, 1981b; Oles, 1982; Oles, 1985] and extended to programming logics in [Tennent, 1986; Tennent, 1990]. For discussion of possible-world semantics of local variables, see [O’Hearn and Tennent, 1992; Lent, 1992; O’Hearn and Tennent, 1993].

The use of non-interference and good-variable formulas to allow Hoare-like reasoning about programs in Algol-like languages with procedures is from [Reynolds, 1981a; Reynolds, 1982]. The interpretation of non-interference formulas presented here is a simplified version of the interpretation given in [O’Hearn, 1990; O’Hearn and Tennent, 1991], which evolved from [Tennent, 1990]. The treatment of block expressions is from [Tennent and Tobin, 1991].

## References

- [Abramsky, 1983a] S. Abramsky. Experiments, powerdomains, and fully abstract models for applicative multiprogramming. In Karpinsky [1983], pages 1–13.
- [Abramsky, 1983b] S. Abramsky. On semantic foundations for applicative multiprogramming. In Diaz [1983], pages 1–14.
- [Abramsky, 1987] S. Abramsky. Domain theory in logical form. In LICS [1987], pages 47–53.
- [Alagić and Arbib, 1978] S. Alagić and M. A. Arbib. *The Design of Well-Structured and Correct Programs*. Springer-Verlag, New York, 1978.

- [Apt, 1981] K. R. Apt. Ten years of Hoare's logic: a survey—part I. *Trans. on Programming Languages and Systems*, 3(4):431–483, 1981.
- [Back, 1983] R. J. Back. A continuous semantics for unbounded nondeterminism. *Theoretical Computer Science*, 23:187–210, 1983.
- [Backhouse, 1986] R. C. Backhouse. *Program Construction and Verification*. Prentice-Hall International, London, 1986.
- [Barendregt, (revised edition) 1984] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, (revised edition) 1984.
- [Barendregt, 1992] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, Oxford, England, 1992.
- [Bekič, 1969] H. Bekič. Definable operations in general algebras and the theory of automata and flowcharts. Technical report, IBM Laboratory, Vienna, 1969. Also, pages 30–55 in C. B. Jones, editor, *Programming Languages and their Definition*, volume 177 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1984.
- [Bergstra and Tucker, 1982] J. A. Bergstra and J. V. Tucker. Expressiveness and the completeness of Hoare's logic. *J. Comput. Sys. Sci.*, 25:276–84, 1982.
- [Bergstra et al., 1982] J. A. Bergstra, J. Tiuryn, and J. Tucker. Floyd's principle, correctness theories, and program equivalence. *Theoretical Computer Science*, 17:129–147, 1982.
- [Berry et al., 1985] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In Nivat and Reynolds [1985], pages 89–132.
- [Brookes, 1985] S. D. Brookes. A fully abstract semantics and a proof system for an ALGOL-like language with sharing. In A. Melton, editor, *Mathematical Foundations of Programming Semantics*, volume 239 of *Lecture Notes in Computer Science*, pages 59–100, Manhattan, Kansas, 1985. Springer-Verlag, Berlin.
- [Carnap, 1947] R. Carnap. *Meaning and Necessity, A Study in Semantics and Modal Logic*. University of Chicago Press, Chicago, 1947.
- [Cartwright and Felleisen, 1992] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *Conf. Record 19th ACM Symp. on Principles of Programming Languages*, pages 328–342, Albuquerque, New Mexico, 1992. ACM, New York.
- [Church, 1940] A. Church. A formulation of the simple theory of types. *J. Symb. Logic*, 5:56–68, 1940.
- [Clarke, 1984] E. M. Clarke, Jr. The characterization problem for Hoare

- logics. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 89–106. Prentice-Hall International, 1984.
- [Clint and Hoare, 1972] M. Clint and C. A. R. Hoare. Program proving: jumps and functions. *Acta Informatica*, 1:214–224, 1972.
- [Connelly, 1990] R. H. Connelly. *A Comparison of Semantic Domains for Interleaving*. Ph.D. thesis, Syracuse University, Syracuse, 1990. Technical report SU-CIS-90-24, School of Computer and Information Science, Syracuse University.
- [Cook, 1978] S. A. Cook. Soundness and completeness of an axiomatic system for program verification. *SIAM J. on Computing*, 7:70–90, 1978.
- [Coppo and Dezani, 1978] M. Coppo and M. Dezani. A new type assignment for  $\lambda$ -terms. *Archiv. Math. Logik*, 19:139–156, 1978.
- [Cousot, 1990] P. Cousot. Methods and logics for proving programs. In van Leeuwen [1990], pages 841–994.
- [Curry and Feys, 1958] H. B. Curry and R. Feys. *Combinatory Logic I*. North-Holland, Amsterdam, 1958.
- [de Bakker and Meertens, 1975] J. W. de Bakker and L. G. L. T. Meertens. On the completeness of the inductive assertion method. *J. Comput. Sys. Sci.*, 11:323–357, 1975.
- [de Bakker et al., 1980] J. W. de Bakker, A. de Bruin, and J. I. Zucker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, London, 1980.
- [de Bruin, 1980] A. de Bruin. Goto statements. In J. W. de Bakker et al. [1980], pages 401–443.
- [Diaz, 1983] J. Diaz, editor. *Proc. 10th Int. Colloq. on Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1983.
- [Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N. J., 1976.
- [Donahue, 1977] J. Donahue. Locations considered unnecessary. *Acta Informatica*, 8:221–242, 1977.
- [Dummett, 1977] M. Dummett. *Elements of Intuitionism*. Oxford University Press, 1977.
- [Ebbinghaus et al., 1984] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer, New York, 1984.
- [Felleisen, 1987] M. Felleisen. *The Calculi of Lambda- $\nu$ -CS-Conversion*. Ph.D. thesis, Indiana University, 1987.
- [Felleisen and Friedman, 1989] M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.



- [Floyd, 1967] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, R.I., 1967.
- [Fourman *et al.*, 1992] M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors. *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, England, 1992.
- [Frege, 1892] G. Frege. Über Sinn und Bedeutung. (On sense and meaning). *Zeitschrift für Philosophie und philosophische Kritik*, 100:25–50, 1892. English translations in *Readings in Philosophical Analysis* (eds., H. Feigl and W. Sellars), pp. 85–102, Appleton-Century-Crofts, New York (1949), and *Translations from the Philosophical Writings of Gottlob Frege* (eds., P. T. Geach and M. Black), pp. 56–78, Blackwell, Oxford (1960).
- [Friedman, 1975] H. Friedman. Equality between functionals. In R. Parikh, editor, *Proceedings of the Logic Colloquium '73*, volume 453 of *Lecture Notes in Mathematics*, pages 22–37. Springer-Verlag, Berlin, 1975.
- [Goldblatt, 1979] R. Goldblatt. *Topoi, The Categorical Analysis of Logic*. North-Holland, Amsterdam, 1979.
- [Goldblatt, 1982] R. Goldblatt. *Axiomatising the Logic of Computer Programming*, volume 193 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1982.
- [Gordon *et al.*, 1979] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF, A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.
- [Gordon, 1979] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.
- [Greif and Meyer, 1981] I. Greif and A. R. Meyer. Specifying the semantics of **while** programs: a tutorial and critique of a paper by Hoare and Lauer. *ACM Trans. Programming Languages and Systems*, 3(4):484–507, 1981.
- [Gunter, 1992] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, Mass., and London, England, 1992.
- [Gunter and Scott, 1990] C. A. Gunter and D. S. Scott. Semantic domains. In van Leeuwen [1990], pages 633–674.
- [Halpern *et al.*, 1984] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Conf. Record 11th ACM Symp. on Principles of Programming Languages*, pages 245–257, Austin, Texas, 1984. ACM, New York.
- [Hennessy, 1990] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*.



John Wiley, Chichester, England, 1990.

- [Hennessy and Plotkin, 1979] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In J. Bečvář, editor, *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer-Verlag, Berlin, 1979.
- [Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580 and 583, 1969.
- [Hoare and Lauer, 1974] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3(2):135–153, 1974.
- [Hortalá-González et al., 1988] M. Hortalá-González, F. Lucio-Carrasco, and M. Rodríguez-Artalejo. Some general incompleteness results for partial correctness logics. *Information and Computation*, 79(1):22–42, 1988.
- [Janssen, 1986] T. M. V. Janssen. *Foundations and Applications of Montague Grammar, part I*, volume 19 of *CWI Tracts*. Center for Mathematics and Computer Science, Amsterdam, 1986.
- [Karpinsky, 1983] M. Karpinsky, editor. *Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1983.
- [Kripke, 1965] S. A. Kripke. Semantical analysis of intuitionistic logic I. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, Amsterdam, 1965.
- [Landin, 1964] P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964.
- [Landin, 1966] P. J. Landin. The next 700 programming languages. *Comm. ACM*, 9(3):157–166, 1966.
- [Lauer, 1971] P. Lauer. Consistent formal theories of the semantics of programming languages. Technical Report TR 25.121, IBM Laboratory, Vienna, Austria, 1971.
- [Lehmann, 1976] D. J. Lehmann. Categories for fixpoint semantics. In *Proc. 17th IEEE Symposium on Foundations of Computer Science*, pages 122–126, 1976.
- [Lehmann and Smyth, 1981] D. J. Lehmann and M. B. Smyth. Algebraic specification of data types: a synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.
- [Leivant, 1985a] D. Leivant. Logical and mathematical reasoning about imperative programs. In *Conf. Record 12th ACM Symp. on Principles of Programming Languages*, pages 132–140, New Orleans, Louisiana, 1985. ACM, New York.

- [Leivant, 1985b] D. Leivant. Partial-correctness theories as first-order theories. In R. Parikh, editor, *Logics of Programs 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 190–195, Brooklyn, N.Y., 1985. Springer-Verlag, Berlin.
- [Leivant and Fernando, 1987] D. Leivant and T. Fernando. Skinny and fleshy failures of relative completeness. In *Conf. Record 14th ACM Symp. on Principles of Programming Languages*, pages 246–252, Munich, West Germany, 1987. ACM, New York.
- [Lent, 1992] A. F. Lent. The category of functors from state shapes to bottomless CPOs is adequate for block structure. Master's thesis, Massachusetts Institute of Technology, 1992.
- [LICS, 1987] *Proceedings, Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1987.
- [Loeckx and Sieber, 1984] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley and Sons, and B. G. Teubner, Stuttgart, 1984.
- [Lucas and Walk, 1969] P. Lucas and K. Walk. On the formal description of PL/I. In *Annual Review in Automatic Programming*, vol. 6, pages 105–152. Pergammon Press, London, 1969.
- [Manes and Arbib, 1986] E. G. Manes and M. A. Arbib. *Algebraic approaches to program semantics*. Springer-Verlag, New York, 1986.
- [Mason and Talcott, 1991] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
- [Mason and Talcott, 1992] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.
- [Meyer, 1986] A. R. Meyer. Floyd–Hoare logic defines semantics: preliminary version. In *Proceedings, Symposium on Logic in Computer Science*, pages 44–48, Cambridge, Mass., 1986. IEEE Computer Society Press.
- [Meyer and Cosmadakis, 1988] A. R. Meyer and S. S. Cosmadakis. Semantical paradigms: notes for an invited lecture. In *Proceedings, 3rd Annual Symposium on Logic in Computer Science*, pages 236–53, Edinburgh, Scotland, 1988. IEEE Computer Society Press.
- [Meyer and Halpern, 1982] A. R. Meyer and J. Y. Halpern. Axiomatic definitions of programming languages: a theoretical assessment. *J. ACM*, 29:555–576, 1982.
- [Meyer and Sieber, 1988] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conf. Record 15th ACM Symp. on Principles of Programming Languages*, pages 191–203, San Diego, California, 1988. ACM, New York.
- [Milne and Strachey, 1976] R. E. Milne and C. Strachey. *A Theory of Pro-*

*gramming Language Semantics*. Chapman and Hall, London, and Wiley, New York, 1976.

- [Milner, 1972] R. Milner. Implementation and applications of Scott's logic for computable functions. In *Proving Assertions About Programs*, pages 1–6, Las Cruces, New Mexico, 1972. ACM, New York. *SIGPLAN Notices*, 7(1), 1972.
- [Milner, 1973] R. Milner. Models of LCF. Technical Report STAN-CS-73-332, Dept. of Computer Science, Stanford University, Stanford, California, 1973. Also in K. R. Apt and J. W. de Bakker, editors, *Foundations of Computer Science II, part 2*, pages 49–63, Mathematical Centre Tracts **82**, Mathematical Centre, Amsterdam, 1976.
- [Milner, 1975] R. Milner. Processes: a mathematical model of computing agents. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 157–174. North-Holland, Amsterdam, 1975.
- [Milner, 1976] R. Milner. Program semantics and mechanized proof. In K. R. Apt and J. W. de Bakker, editors, *Foundations of Computer Science II, part 2*, pages 3–44. Mathematical Centre Tracts **82**, Mathematical Centre, Amsterdam, 1976.
- [Milner, 1977] R. Milner. Fully abstract models of typed  $\lambda$ -calculi. *Theoretical Computer Science*, 4:1–22, 1977.
- [Milner et al., 1975] R. Milner, L. Morris, and M. Newey. A logic for computable functions with reflexive and polymorphic types. In G. Huet and G. Kahn, editors, *Proving and Improving Programs*, pages 371–394, Arc et Senans, France, 1975. INRIA, Rocquencourt, France.
- [Mitchell, 1990] J. C. Mitchell. Type systems for programming languages. In van Leeuwen [1990], pages 365–458.
- [Moggi, 1988] E. Moggi. *The Partial Lambda Calculus*. Ph.D. thesis, University of Edinburgh, 1988.
- [Moggi, 1989] E. Moggi. Computational lambda-calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, 1989. IEEE Computer Society Press.
- [Mosses, 1992] P. D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1992.
- [Mulmuley, 1987] K. Mulmuley. *Full Abstraction and Semantic Equivalence*. The MIT Press, Cambridge, Mass., 1987.
- [Naur et al., 1963] P. Naur, J. W. Backus, et al. Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6(1):1–17, 1963.
- [Nielson and Nielson, 1992] H. R. Nielson and F. Nielson. *Semantics with Applications, a Formal Introduction*. Wiley, Chichester, England, 1992.

- [Nivat and Reynolds, 1985] M. Nivat and J. C. Reynolds, editors. *Algebraic Methods in Semantics*. Cambridge University Press, Cambridge, England, 1985.
- [O'Hearn, 1990] P. W. O'Hearn. *The Semantics of Non-Interference: A Natural Approach*. Ph.D. thesis, Queen's University, Kingston, Canada, 1990.
- [O'Hearn, 1991] P. W. O'Hearn. Linear logic and interference control. In D. H. Pitt et al., editors, *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 74–93, Paris, France, September 1991. Springer-Verlag, Berlin.
- [O'Hearn, 1993] P. W. O'Hearn. A model for syntactic control of interference. To appear in *Mathematical Structures in Computer Science*, 3:435–465, 1993.
- [O'Hearn and Tennent, 1991] P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, part 2. *Information and Computation*, 107:25–57, 1993.
- [O'Hearn and Tennent, 1992] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In Fourman et al. [1992], pages 217–238.
- [O'Hearn and Tennent, 1993] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Conf. Record 20th ACM Symp. on Principles of Programming Languages*, pages 171–184, Charleston, South Carolina, 1993. ACM, New York.
- [Oles, 1982] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.
- [Oles, 1985] F. J. Oles. Type algebras, functor categories and block structure. In Nivat and Reynolds [1985], pages 543–573.
- [Oles, 1987] F. J. Oles. Lambda calculi with implicit type conversions. Technical Report RC 13245, IBM Research, T. J. Watson Research Center, Yorktown Heights, N.Y., 1987.
- [Park, 1969] D. Park. Fixpoint induction and proofs of program properties. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 59–78. Edinburgh University Press, 1969.
- [Pasztor, 1986] A. Pasztor. Non-standard algorithmic and dynamic logic. *J. Symbolic Computation*, 2:59–81, 1986.
- [Paulson, 1987] L. C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1987.
- [Pitts, 1991] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher-Order Workshop*, volume 283 of *Workshops in Computing*, Banff, 1990, 1991. Springer-Verlag, Berlin.



- [Plotkin, 1975] G. D. Plotkin. Call-by-name, call-by-value, and the  $\lambda$ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [Plotkin, 1976] G. D. Plotkin. A powerdomain construction. *SIAM J. on Computing*, 5:452–487, 1976.
- [Plotkin, 1977] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
- [Plotkin, 1978] G. D. Plotkin. *The category of complete partial orders: a tool for making meanings*. Lecture notes for the Summer School on Foundations of Artificial Intelligence and Computer Science, Pisa, June 1978.
- [Plotkin, 1979] G. D. Plotkin. Dijkstra's predicate transformers and Smyth's powerdomains. In D. Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 527–553, Copenhagen, Denmark, 1979. Springer-Verlag, Berlin.
- [Plotkin, 1981] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, Aarhus, Denmark, 1981.
- [Plotkin, 1982] G. D. Plotkin. A powerdomain for countable non-determinism. In *Proc. 9th Int. Colloq. on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 418–428. Springer-Verlag, Berlin, 1982.
- [Plotkin, 1985] G. D. Plotkin. *Types and partial functions*. Lecture notes, Computer Science Department, University of Edinburgh, 1985.
- [Reynolds, 1972] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, 1972.
- [Reynolds, 1978] J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symp. on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, 1978. ACM, New York.
- [Reynolds, 1980] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258, Aarhus, Denmark, 1980. Springer-Verlag, Berlin.
- [Reynolds, 1981a] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.
- [Reynolds, 1981b] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.
- [Reynolds, 1982] J. C. Reynolds. Idealized Algol and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, Cambridge, England, 1982.
- [Reynolds, 1985] J. C. Reynolds. Three approaches to type structure. In



- H. Ehrig, C. Floyd, M. Nivat, and J. W. Thatcher, editors, *Mathematical Foundations of Software Development*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138, Berlin, 1985. Springer-Verlag, Berlin.
- [Reynolds, 1987] J. C. Reynolds. Conjunctive types and Algol-like languages (abstract of invited lecture). In *LICS [1987]*, page 119.
- [Reynolds, 1988] J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Computer Science, Carnegie Mellon University, Pittsburgh, 1988.
- [Reynolds, 1989] J. C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722, Stresa, Italy, 1989. Springer-Verlag, Berlin.
- [Reynolds, 1991] J. C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Sendai, Japan, 1991. Springer-Verlag, Berlin.
- [Riecke, 1990] J. G. Riecke. A complete and decidable proof system for call-by-value equalities. In M. S. Paterson, editor, *Proc. 17th Int. Colloq. on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 20–31. Springer-Verlag, Berlin, 1990.
- [Robinson, 1987] E. P. Robinson. Logical aspects of denotational semantics. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, number 283 in *Lecture Notes in Computer Science*, pages 238–253. Springer-Verlag, 1987.
- [Rosolini, 1986] G. Rosolini. *Continuity and Effectiveness in Topoi*. D.Phil. thesis, Oxford University, 1986.
- [Schmidt, 1986] D. A. Schmidt. *Denotational Semantics, A Methodology for Language Development*. Allyn and Bacon, Newton, Massachusetts, 1986. Reprinted by W. C. Brown, Dubuque, Iowa (1988).
- [Schönfinkel, 1924] M. Schönfinkel. Über die Bausteine der Mathematischen Logik. *Math. Annalen*, 92:305–316, 1924.
- [Schwarz, 1974] J. S. Schwarz. *Semantics of Partial Correctness Formalisms*. Ph.D. thesis, Syracuse University, 1974.
- [Scott, 1969] D. S. Scott. Privately circulated memo, Oxford University, October 1969.
- [Scott, 1970] D. S. Scott. Outline of a mathematical theory of computation. In *Proc. 4th Annual Princeton Conf. on Information Sciences and Systems*, pages 169–176, Princeton, 1970. Also technical monograph PRG-2, Programming Research Group, University of Oxford, Oxford.
- [Scott, 1972a] D. S. Scott. Mathematical concepts in programming language semantics. In *Proc. 1972 Spring Joint Computer Conference*, pages

- 225–34. AFIPS Press, Montvale, N.J., 1972.
- [Scott, 1972b] D. S. Scott. Models for various type-free calculi. In P. Suppes et al., editors, *Logic, Methodology, and the Philosophy of Science, IV*, pages 157–187. North-Holland, Amsterdam, Bucharest, 1972.
- [Scott and Strachey, 1971] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn Press, New York, 1971. Also technical monograph PRG-6, Programming Research Group, University of Oxford, Oxford.
- [Sieber, 1992] K. Sieber. Reasoning about sequential functions via logical relations. In Fourman et al. [1992], pages 258–269.
- [Smyth, 1978] M. B. Smyth. Powerdomains. *J. of Computer and System Sciences*, 16:23–36, 1978.
- [Smyth, 1983] M. B. Smyth. Power domains and predicate transformers: a topological view. In Diaz [1983], pages 662–676.
- [Smyth and Plotkin, 1982] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. on Computing*, 11(4):761–783, 1982.
- [Stoughton, 1988a] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman, London, and Wiley, New York, 1988.
- [Stoughton, 1988b] A. Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–25, 1988.
- [Stoy, 1977] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, and London, England, 1977.
- [Strachey, 1972] C. Strachey. The varieties of programming language. In *Proceedings of the International Computing Symposium*, pages 222–233. Cini Foundation, Venice, 1972. Also technical monograph PRG-10, Programming Research Group, University of Oxford, Oxford.
- [Strachey and Wadsworth, 1974] C. Strachey and C. P. Wadsworth. Continuations: a mathematical semantics for handling full jumps. Technical monograph PRG-11, Programming Research Group, University of Oxford, Oxford, 1974.
- [Tarski, 1956] A. Tarski. *Logic, Semantics, and Meta-Mathematics*. Oxford University Press, Oxford, 1956.
- [Tennent, 1981] R. D. Tennent. *Principles of Programming Languages*. International Series in Computer Science. Prentice-Hall International, 1981.
- [Tennent, 1983] R. D. Tennent. Semantics of interference control. *Theoretical Computer Science*, 27:297–310, 1983.

- [Tennent, 1986] R. D. Tennent. Functor-category semantics of programming languages and logics. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 206–224, Guildford, U.K., 1986. Springer-Verlag, Berlin.
- [Tennent, 1987a] R. D. Tennent. A note on undefined expression values in programming logics. *Inf. Proc. Letters*, 24:331–333, 1987.
- [Tennent, 1987b] R. D. Tennent. Quantification in Algol-like languages. *Inf. Proc. Letters*, 25:133–137, 1987.
- [Tennent, 1989] R. D. Tennent. Elementary data structures in Algol-like languages. *Science of Computer Programming*, 13:73–110, 1989.
- [Tennent, 1990] R. D. Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, 1990.
- [Tennent and Tobin, 1991] R. D. Tennent and J. K. Tobin. Continuations in possible-world semantics. *Theoretical Computer Science*, 85(2):283–303, 1991.
- [Tucker and Zucker, 1988] J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, volume 6 of *CWI Monographs*. North-Holland, Amsterdam, 1988.
- [van Dalen, 1983] D. van Dalen. *Logic and Structure*. Springer, Berlin, second edition, 1983.
- [van Leeuwen, 1990] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B. Elsevier, Amsterdam, and The MIT Press, Cambridge, Mass., 1990.
- [Wadsworth, 1976] C. Wadsworth. The relation between computational and denotational properties for Scott's  $D_\infty$  models of the lambda calculus. *SIAM J. on Computing*, 5:488–521, 1976.
- [Wadsworth, 1978] C. Wadsworth. *Elementary Domains*. Lecture notes, Dept. of Computer Science, University of Edinburgh, 1978.
- [Wand, 1978] M. Wand. A new incompleteness result for Hoare's system. *J. ACM*, 25:168–175, 1978.
- [Wand, 1979] M. Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.
- [Winskel, 1983] G. Winskel. A note on powerdomains and modality. In Karpinsky [1983], pages 505–514.
- [Winskel, 1993] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, Cambridge, Mass., and London, England, 1993.

# Algebraic Semantics

Eric G. Wagner

---

## Contents

1	Introduction and motivation . . . . .	324
	1.1 General remarks . . . . .	324
	1.2 Some motivating examples . . . . .	326
	1.3 A notational ‘crisis’ . . . . .	332
2	Algebraic theories, definitions and examples . . . . .	332
	2.1 General remarks . . . . .	332
	2.2 Basic definitions . . . . .	332
	2.3 Algebraic theories as categories . . . . .	336
	2.4 Examples of algebraic theories . . . . .	337
	2.5 Notations and basic identities . . . . .	341
3	Ordered theories . . . . .	343
	3.1 Definition of ordered and continuous theories . . . . .	343
	3.2 Examples of ordered theories . . . . .	344
4	Theories with iteration operators . . . . .	348
	4.1 Iteration operators . . . . .	348
	4.2 Iteration, rational, and iterative theories . . . . .	349
	4.3 The $\nabla$ -operator . . . . .	356
	4.4 Iteration operators and flowcharts . . . . .	356
	4.5 Interpretations of flowcharts . . . . .	364
5	More about iteration operators . . . . .	366
	5.1 Relationships between iteration, rational, and iterative theories . . . . .	366
	5.2 Some identities for iteration operators . . . . .	372
6	Iteration closure, and normal form theorems . . . . .	374
7	Free theories and Herbrand interpretations . . . . .	378
	7.1 Free theories . . . . .	379
	7.2 The general case . . . . .	380
8	Recursive hierarchies . . . . .	386



# 1 Introduction and motivation

## 1.1 General remarks

The term ‘algebraic semantics’ as used in this chapter, refers to certain approaches to the study of iteration and recursion using either universal algebra or the categorical approach to universal algebra based on Lawvere’s ‘algebraic theories’. The aim of this work is to provide a unified theory of iteration and recursion which covers the common properties of all examples of iteration and recursion. Thus, as we will show, algebraic semantics provides a uniform treatment of the theory of such seemingly diverse subjects as context-free grammars, flowcharts, recursion schemata, and recursively defined domains.

The first four sections of the chapter provide a basic introduction to the subject, the final four sections take a deeper look into selected topics. We begin by dedicating most of this section to looking at three motivating examples: flowcharts, recursion equations, and context-free grammars. The aim is to give the reader some intuitive feel for the common features of these seemingly diverse examples. In particular, in all three examples the ‘meaning’ is given as a fixpoint. That is, speaking very loosely, the meaning of a flowchart (recursion equation, grammar)  $\alpha$ , is an object  $\alpha^\dagger$  such that  $\alpha \bullet \alpha^\dagger = \alpha^\dagger$  where ‘ $\bullet$ ’ is some kind of composition operation. The question then is, how can we make this precise?

The approach taken in this chapter, the approach that allows us to treat these and other examples uniformly is based on the concept of a Lawvere algebraic theory, [Lawvere, 1963]. Depending on one’s viewpoint, a Lawvere algebraic theory is either a special kind of many-sorted algebra or a special kind of category. While Lawvere originally developed algebraic theories to deal with questions in universal algebra they have a history of applications in computer science dating back to work in automata theory by Eilenberg and Wright [1967]. They have also been used in the theory of data type specification [Wagner and Ehrig, 1987; Bloom and Wagner, 1985] and in the semantics of programming languages [Goguen and Burstall, 1977; Goguen and Burstall, 1981; Thatcher *et al.*, 1981], etc. Because the concept of algebraic theory has broad applications we try to give a somewhat fuller treatment than is strictly needed here. While our treatment employs algebraic theories it is important to note that many of the same ideas can be, and were, developed by Guessarian, Nivat and others using just universal algebra. In particular, material in Section 7 was originally developed by Guessarian in such a universal algebra framework, see [Guessarian, 1981].

Section 2 defines ‘ordinary’ algebraic theories, and gives a variety of examples. For reasons explained at the end of this introduction, the definition given there is the dual of that in [Lawvere, 1963]. For many applications, including those of algebraic semantics, we need algebraic theories with additional structure such as ordering and/or an iteration operation.



In Section 3 we introduce ordered and  $\omega$ -continuous algebraic theories, together with further examples. Then, in Section 4, we come to the heart of the matter: theories with an iteration, or fixpoint, operator,  $\dagger$ . We present three different, but closely related, ways to define algebraic theories with iteration operators. A preiteration theory is just an algebraic theory with an additional operation  $\dagger$ . One example of such a preiteration theory is the theory of labeled trees which are unfoldings of finite flowchart schemes (see below, Section 4). The class of iteration theories, introduced by Bloom, Elgot and Wright in [1980a; 1980b] is defined as the class of all preiteration theories in which the iteration operation satisfies all the equations true in this tree theory. These equations were axiomatized first by Ésik in [Ésik, 1980], and an easy algorithm exists to determine the validity of any given equation. Since algebraic theories are also defined equationally this means that we can exploit the results of universal algebra in our study of iteration theories. The second approach, due to ADJ, uses special ordered algebraic theories called *rational theories*, to provide a general treatment of the notion of a least fixpoint. All rational theories are iteration theories but the converse does not hold. However, most of the ‘familiar’ iteration theories are rational theories, and the order-theoretic properties of rational theories lead to a number of special results. The final approach, due to Elgot, studies theories, called *iterative theories* in which the fixpoints are unique [Elgot, 1975]. We show, in some detail, how the examples of Section 1 can be redone rigorously in this framework in terms of an abstract concept of ‘flowcharts’ in an algebraic theory. We end the section with a discussion of how interpretations of ‘flowcharts’ can be viewed naturally as morphisms between algebraic theories.

Section 5 provides a comparison between iteration, rational and iterative theories. In addition it presents a collection of basic identities for iteration operators, which will be needed in the remaining sections. These identities also illustrate the advantages of working within such a general framework over having to separately prove these identities for each example.

Section 6 deals with the concept of iterative closure. We show that the ‘flowcharts written in’ any (not necessarily iteration) subtheory of an iteration (or rational) theory form an iteration (rational) theory. We then use this result to prove a generalization of the Chomsky normal form theorem that underlies many syntactic forms of programming languages, in addition to grammars.

In Section 7 we look at Herbrand models for classes of interpretations of ‘flowcharts’, in iteration, rational and  $\omega$ -continuous algebraic theories.

Finally in Section 8 we investigate the construction of natural hierarchies of iteration theories. This provides a general treatment of well-known hierarchies such as

regular languages, context-free languages, indexed languages, ...

and hierarchies of higher- and higher-level functional languages.

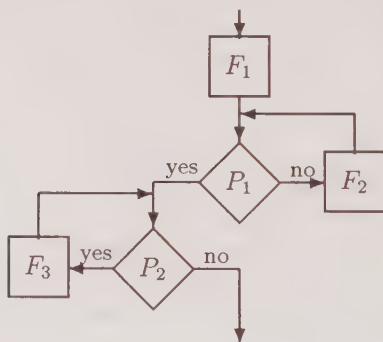
While the term 'algebraic semantics' is frequently used to refer to forms of denotational semantics employing algebraic concepts (e.g., see [Gordon, 1979]) or to the algebraic specification of data types, [Goguen *et al.*, 1978], the more restricted usage employed here goes back at least to Guessarian's book [Guessarian, 1981].

## 1.2 Some motivating examples

We begin by giving three informal examples of recursive definitions which we will use in this section to provide intuitive motivation for the chapter as a whole, and which we will come back to repeatedly to illustrate steps in our development. The first example is a flowchart, the second example is a set of recursion equations, and the third is a context-free grammar.

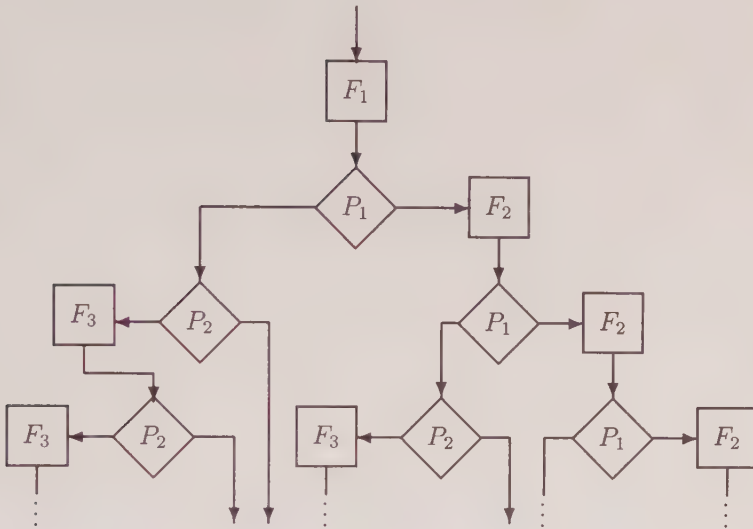
### Example 1.2.1.

Consider the following flowchart:



If we interpret the symbols in the flowchart by, say, viewing each  $F_i$  as a state transformation on some fixed set,  $S$ , of states, and viewing each  $P_i$  as a binary predicate on  $S$ , then, intuitively, the flowchart as a whole has a meaning as a, possibly partial, function from  $S$  to  $S$  in accordance with the usual informal interpretation of flowcharts.

The flowchart, as shown, is uninterpreted, that is, we have not said what actual functions are executed by the indicated boxes and diamonds. But even without giving an interpretation to the boxes, we know, intuitively, that for any interpretation, the function realized by the flowchart will be the same as that of the infinite tree-like (i.e., loop-free) flowchart suggested by the following picture.



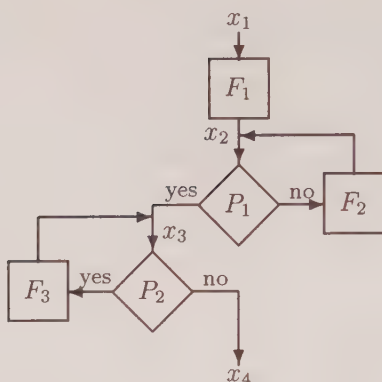
What we have said so far is fine intuitively, but how can we characterize the above infinite tree more precisely? For that matter, how can we make the notions of a flowchart, the interpretation of a flowchart, and the unwinding of a flowchart, more precise? Can we answer these questions in such a way that the answers hold not only for flowcharts but also for other examples of recursion such as recursion equations, monadic schemes, regular grammars, and context-free grammars? This chapter, of course, is dedicated to giving a positive, and detailed, answer to these questions. But before approaching the general answer let us motivate it by taking a closer, and slightly more formal, look at the above special case of uninterpreted flowcharts. Essentially the same ideas will be presented more formally in Section 4. For technical reasons, the two treatments do not quite match.

We will do everything in terms tuples of tree-like flowcharts with labelled outputs. That is, with flowcharts that have no loops and in which each output has a label from the set  $X = \{x_1, x_2, \dots, x_n, \dots\}$ .

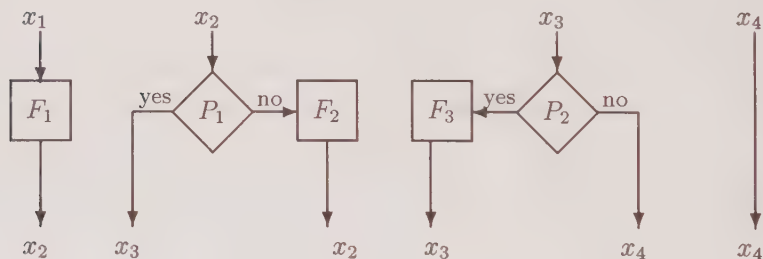
A flowchart with loops can be represented in terms of tree-like flowcharts. The idea is to 'cut the loops' in the flowchart so as to reduce it to a tuple of trees. We illustrate the idea using the flowchart given above.

We begin by choosing points on the flowchart in such a manner that cutting it at the marks will break it into a tuple of trees. Next we mark the input, the output, and above chosen points in the flowchart with distinct labels from an initial segment of  $X_n$  of the set  $X$ , with the input having label  $x_1$  and the output having label  $x_n$ .

Here is one such marking for the above flowchart, using the set of marks  $X_4 = \{x_1, x_2, x_3, x_4\}$ :

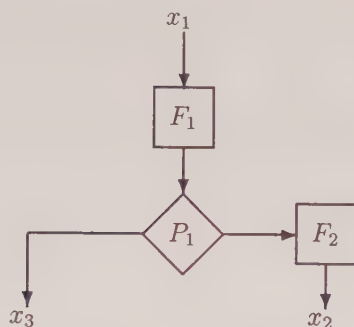


After we break the flowchart into tree-like flowcharts we label their inputs (roots) and outputs (leaves) with the marks from the corresponding points in the original flowchart. Here is the 4-tuple of trees corresponding to the above flowchart:

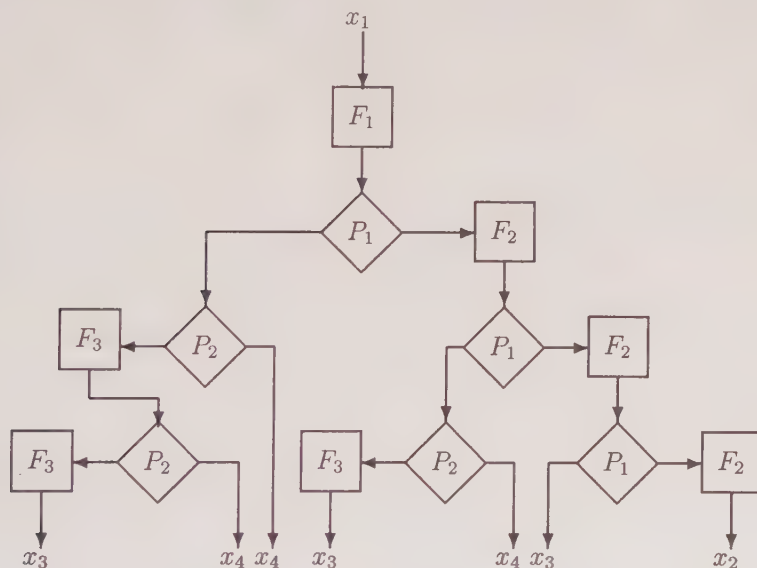


We will work with just one operation on these tuples, namely a *composition operation*,  $\bullet$ . Given an  $n$ -tuple  $T = \langle t_1, \dots, t_n \rangle$  of tree-like flowcharts with output labels restricted to the set  $X_p = \{x_1, \dots, x_p\}$ , we can compose it with any  $p$ -tuple,  $T' = \langle t'_1, \dots, t'_p \rangle$  of tree-like flowcharts with output labels from the set  $X_q = \{x_1, \dots, x_q\}$ . The result of composing  $T$  and  $T'$ , denoted  $T \bullet T'$  will be the  $n$ -tuple  $\langle t''_1, \dots, t''_n \rangle$  of trees with outputs labels from  $X_q$  such that, for each  $i = 1, \dots, n$ ,  $t''_i$  is the result of simultaneously, for  $j = 1, \dots, p$ , replacing each occurrence of  $x_j$  in  $t_i$  with  $t'_j$ .

Let us write  $T = \langle t_1, t_2, t_3, t_4 \rangle$  to represent the above 4-tuple of trees. Then, for example,  $t_1 \bullet T$  is the tree



and  $((t_1 \bullet T) \bullet T) \bullet T$  is the tree



We can use the representation of a flowchart as a tuple, and the composition  $\bullet$  to characterize the infinite flowchart corresponding to the unwinding of the flowchart.

Letting  $T$  be the 4-tuple given above, the first characterization uses the fact that  $T$  has a fixed point  $T^\dagger$  such that  $T \bullet T^\dagger = T^\dagger$ . That is, there exists a 4-tuple  $T^\dagger = \langle t'_1, t'_2, t'_3, t'_4 \rangle$  of tree-like flowcharts (with  $t'_i$   $i = 1, 2, 3$  being infinite flowcharts) such that

$$T \bullet T^\dagger = \langle t_1 \bullet T^\dagger, t_2 \bullet T^\dagger, t_3 \bullet T^\dagger, t_4 \bullet T^\dagger \rangle = T^\dagger.$$

The desired unwinding is precisely the first component,  $t'_1$ , of  $T^\dagger$ .

In the case of flowcharts we can show that for all tuples  $T$  representing 'interesting' flowcharts the fixpoint  $T^\dagger$  is uniquely defined.





= multiplication, and  $f_3$  = factorial. We can regard the set,  $E$ , of equations given above as an operator on the set of partial functions on the natural numbers:

$$E : [\omega \times \omega \rightarrow \omega] \times [\omega \times \omega \rightarrow \omega] \times [\omega \rightarrow \omega] \rightarrow \\ [\omega \times \omega \rightarrow \omega] \times [\omega \times \omega \rightarrow \omega] \times [\omega \rightarrow \omega]$$

Then the desired solution will be the least fixpoint with respect to the usual ordering on the partial functions.

### Example 1.2.3.

Consider the following context-free grammar,  $G$ , for a fragment of a simple programming language

$$\begin{aligned} \langle \text{st} \rangle &::= (\langle \text{a-id} \rangle := \langle \text{ae} \rangle) \mid (\langle \text{b-id} \rangle := \langle \text{be} \rangle) \mid (\text{if } \langle \text{be} \rangle \text{ then } \langle \text{st} \rangle \text{ else } \langle \text{st} \rangle) \\ &\quad \mid \langle \text{st} \rangle ; \langle \text{st} \rangle \mid (\text{while } \langle \text{be} \rangle \text{ do } \langle \text{st} \rangle) \\ \langle \text{ae} \rangle &::= 0 \mid \langle \text{a-id} \rangle \mid \text{succ}(\langle \text{ae} \rangle) \mid (\langle \text{ae} \rangle + \langle \text{ae} \rangle) \mid (\langle \text{ae} \rangle * \langle \text{ae} \rangle) \\ &\quad \mid (\text{if } \langle \text{be} \rangle \text{ then } \langle \text{ae} \rangle \text{ else } \langle \text{ae} \rangle) \\ \langle \text{be} \rangle &::= \text{true} \mid \text{false} \mid \langle \text{b-id} \rangle \mid \text{is-zero}(\langle \text{ae} \rangle) \mid \neg(\langle \text{be} \rangle) \mid (\langle \text{be} \rangle \wedge \langle \text{be} \rangle) \end{aligned}$$

We take the point of view that the above grammar should be viewed as a triple of equations. That the desired interpretation of the equations are as functions on the set of sets of strings on the alphabet

$$\mathcal{A} = \{ :=, \text{while}, \text{do}, 0, \text{succ}, +, *, \text{if}, \text{then}, \text{else}, \\ \text{true}, \text{false}, \text{is-zero}, \neg, \wedge \}.$$

together with the sets of 'identifiers' corresponding to  $\langle \text{a-id} \rangle$  and  $\langle \text{b-id} \rangle$ . Then the least fixpoint,  $G^\dagger$ , of the set of equations is the sets of Statements, Arithmetic and Boolean expressions. Actually, because the sets corresponding to  $\langle \text{a-id} \rangle$  and  $\langle \text{b-id} \rangle$  are not given, the fixed point,  $G^\dagger$ , is itself a function of two arguments, such that, given sets  $A$  and  $B$  of, respectively, arithmetic and boolean identifiers,  $G^\dagger(A, B)$  is a triple of sets of strings.

The goal of algebraic semantics is to capture, and investigate, common structure that underlies the three examples given above. Our first goal will be to replace the informal idea of tuples of tree-like flowcharts or tuples of equations with an appropriate abstract mathematical construct. The construct we will use here is the Lawvere algebraic theory, a construct which may be defined as either a special kind of many-sorted universal algebra or a special kind of category. We will see that all the above examples can be viewed as such algebraic theories, and that there are many other examples of computer science interest. We will then show how the concept of algebraic theory has been extended, in various ways, to include an iteration operator, the  $\dagger$  of the above examples. We will then investigate the properties and models of theories with iteration operators.

### 1.3 A notational ‘crisis’

In writing this chapter I have run head-on into the notational problems that arise from the conflict of how to write the composition of functions and other morphisms. I have, perforce, followed the convention used throughout this Handbook of writing the composite of morphisms  $f : A \rightarrow B$  and  $g : B \rightarrow C$  as  $g \bullet f : A \rightarrow C$  rather than using the ‘diagrammatic notation’  $f \bullet g$ . Unfortunately, many, if not most, of the references for this chapter that employ algebraic theories use the diagrammatic notation. Furthermore, with the usual definition of algebraic theory, the diagrammatic notation is preferable since for the important case of algebraic theories of terms over a signature, the terms are written in the familiar prefix notation.

My resolution of this notational crisis has been to replace the usual definition of Lawvere categorical theory with its dual (that is, I have turned around all the arrows) and then to follow the Handbook’s convention on composition. This essentially preserves appearances. My axioms and identities look like the ones in the literature, and terms look like ‘terms’. However, what were coproducts in the literature are now products, and so on. Furthermore the arrows in examples now run in the opposite direction than they used to. This, however, is no real problem as one of the unavoidable oddities of the subject is that the arrows in about half the examples have always seemed to be ‘going in the wrong direction’—so all we have done is change which examples look wrong.

## 2 Algebraic theories, definitions and examples

### 2.1 General remarks

This section provides the basic definitions and notations for many-sorted Lawvere algebraic theories, together with some fundamental identities. To illustrate the intuition behind algebraic theories we follow the formal definition of algebraic theory with an informal, pictorial treatment of ‘an algebraic theory of boxes’. More formal examples of algebraic theories are given at the end of the section.

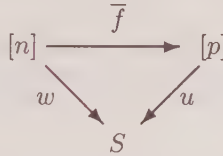
### 2.2 Basic definitions

We define many-sorted Lawvere algebraic theories as a special kind of many-sorted algebra where the sorts are indexed by pairs of strings on an alphabet  $S$ . Such an algebraic theory can also be viewed as a special kind of category where the set of objects is precisely the set of strings on a set  $S$ . In either case we need to start by introducing a precise, mathematical, concept of string.

**Definition 2.2.1.** Let  $\omega$  denote the set of natural numbers,  $\{0, 1, 2, \dots\}$ . Let  $[\omega]$  denote the set of positive natural numbers,  $\{1, 2, 3, \dots\}$ . For any  $n \in \omega$  let  $[n] = \{1, \dots, n\}$ , so  $[0] = \emptyset$ .

Given a set  $S$ , let  $S^n$ , the set of strings on  $S$  of length  $n$ , be the set of all mappings  $w : [n] \rightarrow S$ . Let  $S^*$ , the set of all strings on  $S$ , be the set  $S^* = \bigcup \{S^n \mid n \in \omega\}$ . The unique string  $\lambda : [0] \rightarrow S$  is called the *empty string* on  $S$ . Given a string  $u \in S^*$  let  $|u|$  denote the length of  $u$ , so  $u : [|u|] \rightarrow S$ . Given a set  $S$ , let  $S^n$ , the set of strings on  $S$  of length  $n$ , be the set of all mappings  $w : [n] \rightarrow S$ . Let  $S^*$ , the set of all strings on  $S$ , be the set  $S^* = \bigcup \{S^n \mid n \in \omega\}$ . The unique string  $\lambda : [0] \rightarrow S$  is called the *empty string* on  $S$ . Given a string  $u \in S^*$  let  $|u|$  denote the length of  $u$ , so  $u : [|u|] \rightarrow S$ . Given  $u : [n] \rightarrow S$  we will frequently employ the usual informal notation for strings writing  $u_i$  for  $u(i)$  and  $u = u_1 \cdots u_n$ .

We form a *category of strings on  $S$* ,  $\mathbf{Str}_S$ , with objects  $|\mathbf{Str}_S| = S^*$ , where for  $w, u \in |\mathbf{Str}_S|$ , with  $w : [n] \rightarrow S$  and  $u : [p] \rightarrow S$ , a *string morphism*  $f : w \rightarrow u$  is a triple  $\langle w, \bar{f}, u \rangle$  where  $\bar{f} : [n] \rightarrow [p]$  is a mapping such that  $u \cdot \bar{f} = w$ .



Given strings  $u \in S^n$  and  $v \in S^p$  let  $u + v$  denote the string

$$\begin{aligned}
 u + v &: [n + p] \rightarrow S \\
 i &\mapsto u(i) \text{ if } 1 \leq i \leq n \\
 i &\mapsto v(i - n) \text{ if } n < i \leq n + p
 \end{aligned}$$

Here now is the definition of an  $S$ -sorted algebraic theory as an  $(S^* \times S^*)$ -sorted algebra:

**Definition 2.2.2.** Let  $S$  be a set, then an  $S$ -sorted (Lawvere) algebraic theory is an  $S^* \times S^*$ -sorted algebra with

Carriers:  $T(u, v)$ ,  $u, v \in S^*$ ,

Operations:

$x_i^u \in T(u, u_i)$ , for each  $u = u_1 \cdots u_n \in S^*$ , and  $i$ ,  $1 \leq i \leq n$ .

$\bullet_{u,v,w} : T(u, v) \times T(v, w) \rightarrow T(u, w)$ , for all  $u, v, w \in S^*$ . Given  $\alpha \in T(u, v)$  and  $\beta \in T(v, w)$  we write  $\beta \bullet \alpha$  for  $\bullet_{u,v,w}(\alpha, \beta)$ .

$(, \dots, )_{u,v} : T(u, v_1) \times \cdots \times T(u, v_n) \rightarrow T(u, v)$ , for all  $u \in S^*$ , and all  $v = v_1 \cdots v_n \in S^*$ . Given  $\alpha_i \in T(u, v_i)$  for each  $i \in |v|$ , we write  $(\alpha_1, \dots, \alpha_{|v|})_{u,v}$  for  $(, \dots, )_{u,v}(\alpha_1, \dots, \alpha_{|v|})$ .

Axioms:

(2.2.2.1)  $x_i^v \bullet_{u,v,v_i} (\alpha_1, \dots, \alpha_n)_{u,v} = \alpha_i$  for all  $\alpha_i \in T(u, v_i)$ , where  $1 \leq i \leq n = |v|$ .

(2.2.2.2)  $(x_1^v \bullet_{u,v,v_1} \beta, \dots, x_{|v|}^v \bullet_{u,v,v_{|v|}} \beta)_{u,v} = \beta$  for all  $\beta \in T(u, v)$ .

(2.2.2.3)  $(x_1^u)_{u,u_1} = x_1^u$  for all  $u \in S^*$

(2.2.2.4)  $(\gamma \bullet_{u,v,w} \beta) \bullet_{u,v,x} \alpha = \gamma \bullet_{u,w,x} (\beta \bullet_{u,v,w} \alpha)$  for all  $\alpha \in \mathbf{T}(u,v)$ ,  
 $\beta \in \mathbf{T}(v,w), \gamma \in \mathbf{T}(w,x)$

(2.2.2.5) If  $u \in S^*$ , where  $u = u_1 \cdots u_{|u|}$ , then for all  $v \in S^*$ , and  
 $\gamma : v \rightarrow u$ ,

$$\gamma \bullet_{u,u,v} (x_1^u, x_2^u, \dots, x_{|u|}^u)_{u,u} = \gamma.$$

Elements of  $\mathbf{T}(u,v)$  are called (*theory*) *morphisms* with *source*  $u$  and *target*  $v$ . We often write  $\alpha : u \rightarrow v$  for  $\alpha \in \mathbf{T}(u,v)$ . The operations  $x_i^u$  are often called *distinguished morphisms*, the operations  $\bullet_{u,v,w}$  are called *composition operations*, the operations  $(\dots)_{u,v}$  are called *tupling operations*. Operations  $f : u \rightarrow w$  built up by tupling distinguished morphisms are called *base morphisms*. When there is no ambiguity we write ‘ $\bullet$ ’ rather than ‘ $\bullet_{u,v,w}$ ’, and ‘ $(\dots)$ ’ rather than ‘ $(\dots)_{u,v}$ ’.

If  $S$  is a singleton set, then we say that  $\mathbf{T}$  is a *1-sorted algebraic theory*, and we identify the elements of  $S^*$  with the natural numbers, i.e., where  $S = \{s\}$ , we write  $n$  for  $s^n$ .

The  $S$ -sorted theories form a category,  $\mathbf{Th}_S$ , in which a morphism  $H$  between algebraic theories  $\mathbf{T}_1$  and  $\mathbf{T}_2$  is just a homomorphism between the corresponding  $(S^* \times S^*)$ -sorted algebras. That is,  $H$  is an  $(S^* \times S^*)$ -indexed family

$$H = \langle H_{u,v} : \mathbf{T}_1(u,v) \rightarrow \mathbf{T}_2(u,v) \mid u,v \in S^* \rangle,$$

preserving composition, tupling, and the distinguished morphisms.

Note that care must be taken to distinguish between morphisms in theories (e.g.,  $\alpha \in \mathbf{T}(w,v)$ ) and morphisms between theories (e.g.,  $F : \mathbf{T} \rightarrow \mathbf{T}'$ ).

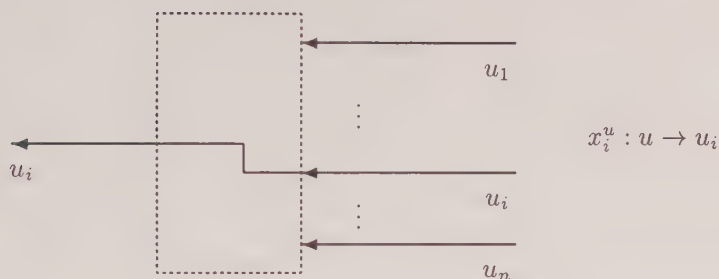
As the use of terms such as ‘morphism’, ‘source’, ‘target’, and ‘composition’ may suggest, an  $S$ -sorted Lawvere algebraic theory  $\mathbf{T}$  can always be viewed as a category. But before looking at this more closely, we want to give an informal way of looking at many-sorted algebraic theories which should provide some useful intuitions in this and following sections. Let  $\mathbf{T}$  be an  $S$ -sorted algebraic theory for some set  $S$ . Then regard a morphism  $\alpha \in \mathbf{T}(u,v)$  as a ‘black box’



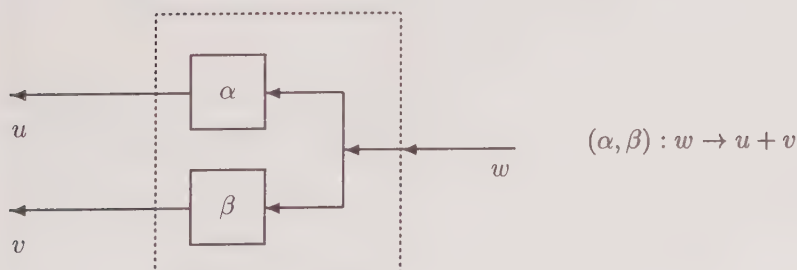
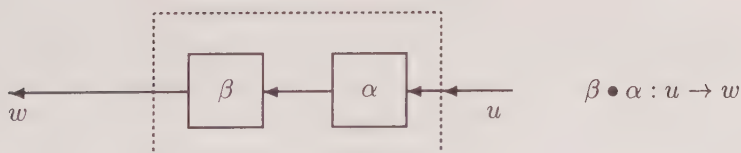
with  $|u|$  inputs of respective sorts  $u_1$  through  $u_{|u|}$ , and  $|v|$  outputs of respective sorts  $v_1$  through  $v_{|v|}$ . Note that the ‘arrows’ on the box go the opposite way to the ‘arrow’ on the morphism  $\alpha : u \rightarrow v$ . This is done so that in pictures corresponding to composites of morphisms, such as that given below for  $\beta \bullet \alpha$ , the boxes appear in the same order.



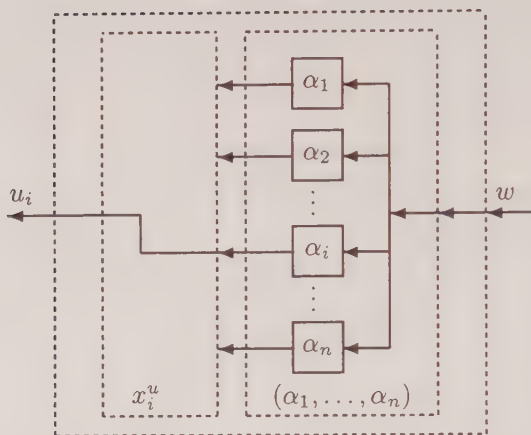
Then we get the following pictorial interpretations for  $x_i^u$ ,  $\beta \bullet \alpha$ , and  $(\alpha, \beta)$ :



i.e., the  $i$ th 'wire' goes through the box, but the others do not.



Such pictures can also be used to explicate the axioms. For example, given  $u = u_1 \dots u_n \in S^*$  and  $\alpha_i : w \rightarrow u_i$ , for  $i = 1, \dots, n$  we see that  $x_i^u \bullet (\alpha_1, \dots, \alpha_n)$  corresponds to the picture



But clearly, under any reasonable interpretation of the  $\alpha_j$ 's this 'circuit' has the same behaviour as the simple 'circuit'



corresponding to  $\alpha_i$ , and this is just what is said by the axiom,

$$x_i^u \bullet (\alpha_1, \dots, \alpha_n) = \alpha_i.$$

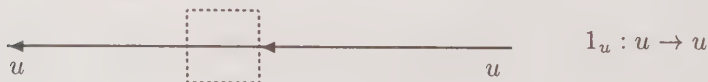
### 2.3 Algebraic theories as categories

The following results show us how to look at an  $S$ -sorted algebraic theory  $\mathbf{T}$  as a category. They also introduce some special morphisms  $1_u : u \rightarrow u$  and  $0_u : \lambda \rightarrow u$  and some useful notation.

**Proposition 2.3.1.** *An  $S$ -sorted algebraic theory  $\mathbf{T}$  forms a category, also denoted by  $\mathbf{T}$ , with objects  $|\mathbf{T}| = S^*$ ; with  $\mathbf{T}(u, v)$  as the set of morphisms from  $u$  to  $v$ ; with composition operation  $\bullet$ ; and, for any  $u = u_1 \cdots u_n \in S^n$ ,  $1_u = (x_1^u, \dots, x_n^u)$  as the identity morphism on  $u$ .*

*Furthermore, if  $u = u_1 \cdots u_n \in S^n$ , then  $\langle u, \langle x_i^u \mid i = 1, \dots, n \rangle \rangle$  is a product in  $\mathbf{T}$ .*

The 'black box' corresponding to  $1_u$  is given by the picture



**Fact 2.3.2.** If  $u \in S^*$ , then there is a unique morphism  $0_u : u \rightarrow \lambda$ , namely  $0_u = ( , \dots, )_{u, \lambda}$ , the ‘empty tuple’.

We can picture  $0_u$  as follows:



## 2.4 Examples of algebraic theories

Here are a number of examples of algebraic theories most of which have computer science motivations.

**Example 2.4.1.** For a given set,  $S$ , of sorts, the base morphisms form an algebraic theory which, in fact, can be identified with the category,  $\mathbf{Str}_S^{op}$ , the dual of the category  $\mathbf{Str}_S$  of strings on  $S$  (see Definition 2.2.1). Given strings  $u : [n] \rightarrow S$  and  $v : [p] \rightarrow S$ , recall that a string morphism  $f : u \rightarrow v$  is determined by a mapping  $\bar{f} : [n] \rightarrow [p]$  such that  $v \bullet \bar{f} = u$ . The base morphism corresponding to  $f$  is the tuple

$$(x_{\bar{f}(1)}^v, x_{\bar{f}(2)}^v, \dots, x_{\bar{f}(n)}^v) : v \rightarrow u$$

of distinguished morphisms,  $x_{\bar{f}(i)}^v : v \rightarrow v_{f(i)} = u_i$ .

**Example 2.4.2.** Given a set  $A$ , let  $\mathbf{Sum}_A$  be the 1-sorted algebraic theory in which, for all  $n, p \in \omega$ ,  $\mathbf{Sum}_A(n, p)$  consists of all partial functions  $f : A \times [p] \rightarrow A \times [n]$  (note the reversal). Identify  $A \times [1]$  with  $A$ . Given  $f_i : A \rightarrow A \times [p]$ ,  $i = 1, \dots, n$ , define the tupling of  $f_1$  through  $f_n$  to be

$$(f_1, \dots, f_n) : A \times [n] \rightarrow A \times [p]$$

$$\langle a, i \rangle \mapsto f_i(a).$$

Given  $i \in [n]$  we define the distinguished morphism

$$x_i^n : A \times [1] \rightarrow A \times [n]$$

$$a \mapsto \langle a, i \rangle.$$

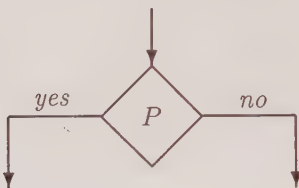
Given  $f \in \mathbf{Sum}_A(n, p)$  and  $g \in \mathbf{Sum}_A(p, q)$  corresponding, respectively, to partial functions  $f : A \times [p] \rightarrow A \times [n]$  and  $g : A \times [q] \rightarrow A \times [p]$ , their composite  $f \bullet g \in \mathbf{Sum}_A(n, q)$  corresponds to the composite  $fg : A \times [q] \rightarrow A \times [n]$ , of the corresponding partial functions. The identity morphism  $1_n : n \rightarrow n$  is just the identity mapping on  $A \times [n]$ .

We can view  $\mathbf{Sum}_A$  as an interpreted theory of flowcharts. For this purpose consider  $A$  to be a *set of states*. Then a morphism  $\alpha \in \mathbf{Sum}_A(n, p)$  can be interpreted as a “black box” with  $p$  input channels, and  $n$  output channels, where one enters the box on some channel  $i \in [p]$  in some state  $a \in A$ , and then, if  $\alpha(\langle a, i \rangle) = \langle a', j \rangle$ , one leaves the box on channel  $j \in [n]$  in state  $a'$ .

The familiar concept of a flowchart built up blocks for state transformations  $F : A \rightarrow A$  represented by boxes



and conditionals (predicates)  $P$  on  $A$  represented as diamonds



fits nicely into this framework. The state transformation is just the appropriate function  $F : A \rightarrow A \times [1]$  and the conditional  $P$  is represented by a function  $\overline{P} : A \rightarrow A \times [2]$  such that

$$\overline{P}(a) = \begin{cases} \langle a, 1 \rangle & \text{if } P(a) \\ \langle a, 2 \rangle & \text{if } \neg P(a). \end{cases}$$

The algebraic theory  $\mathbf{Sum}_A$  gives us a model of interpreted flowcharts, but how do we get a formal counterpart of our informal pictorial algebraic theory of uninterpreted flowcharts? To make this precise we have to first develop a precise concept of theories of finite labeled trees (or expressions). The effort required will not be wasted since the resulting algebraic theories will play an important role in what is yet to come – they will turn out to be the free theories. The free theories in a category of theories are those in which the only identities that hold are the identities which hold in all the theories in the category. We will be looking at many different categories of theories and in almost every case the free theories will be of significant interest.

**Definition 2.4.3.** Let  $S$  be a set. By an  $S$ -sorted signature we mean an  $(S^* \times S)$ -indexed family of sets  $\Sigma = \langle \Sigma_{w,s} \mid w \in S^*, s \in S \rangle$  of disjoint sets. Given  $S$ -sorted signatures  $\Sigma$  and  $\Omega$ , a *signature morphism*  $f : \Sigma \rightarrow \Omega$  is an  $(S^* \times S)$ -indexed family  $f = \langle f_{w,s} : \Sigma_{w,s} \rightarrow \Omega_{w,s} \mid w \in S^*, s \in S \rangle$ , of mappings.

**Definition 2.4.4.** Let  $\Sigma$  be an  $S$ -sorted signature. For each  $n \in [\omega]$  and  $u \in S^n$  let  $X_u = \{x_1^u, \dots, x_n^u\}$  be an  $[n]$ -indexed set (of *variables*) disjoint

from  $\Sigma_{w,s}$  for all  $w \in S^*$  and  $s \in S$ , and disjoint from all other  $X_v$  for  $v \neq u$ . Call  $x_i^u \in X_u$  the *i*th variable of  $X_u$ . Define  $\Sigma \cup X_u$  to be the  $S$ -sorted signature with  $(\Sigma \cup X_u)_{\lambda,s} = \Sigma_{\lambda,s} \cup \{x_i^u \mid u_i = s\}$ , and with  $(\Sigma \cup X_u)_{v,s} = \Sigma_{v,s}$  for  $v \neq u$ .

For each  $w \in S^*$  define the set  $T_\Sigma(X_w)$  of *total finite w-ary S-sorted  $\Sigma$ -trees* as the set of all partial functions  $t : [\omega]^* \rightarrow \Sigma \cup X_w$  such that,

1. for all  $v \in [\omega]^*$ , and  $j \in [\omega]$ , if  $t(vj) = \xi \in (\Sigma \cup X_w)_{v,s}$ , then there exists  $n \geq j$ , and  $u \in S^n$ ,  $q \in S$ , and  $\alpha \in (\Sigma \cup X_w)_{u,q}$ , such that  $u_j = s$  and  $t(v) = \alpha$ .
2. for all  $v \in [\omega]^*$ , if  $t(v) = \alpha \in \Sigma_{u,s}$ , where  $u \in S^n$ , then for each  $i \in [n]$ ,  $t(vi)$  is defined.
3.  $t$  is defined on a finite, nonempty, subset of  $[\omega]^*$ .

A  $w$ -ary tree  $t$  will be said to be of *sort*  $s$  if  $t(\lambda) \in \Sigma_{u,s} \cup \{x_i^w \mid w_i = s\}$  for some  $u \in S^*$ .

**Example 2.4.5.** With  $S$  a set and  $\Sigma$  an  $S$ -sorted signature,  $\mathbf{T}_\Sigma$ , the *free  $S$ -sorted  $\Sigma$ -(generated-)Theory* is defined as follows:

(2.4.5.1) For each  $v \in S^n$ ,  $w \in S^*$ ,  $\mathbf{T}_\Sigma(w, v)$  is the set of all triples  $\langle w, t, v \rangle$  where  $t$  is an  $n$ -tuple  $t = (t_1, \dots, t_n)$  of total, finite  $w$ -ary  $\Sigma$ -trees such that  $t_i$  is of sort  $v_i$  for each  $i \in [n]$ . While defining a morphism  $\alpha = \langle v, t, w \rangle$  in  $\mathbf{T}_\Sigma(w, v)$  as a triple is necessary to ensure that the source and target of  $\alpha$  are uniquely determined, it is very convenient to informally identify  $\alpha$  with the  $n$ -tuple  $t = (t_1, \dots, t_n)$ . We shall employ this informal device in the rest of this definition, and throughout the chapter.

(2.4.5.2) For  $u \in S^n$ ,  $v \in S^p$  and  $w \in S^*$ , and for  $t = (t_1, \dots, t_n) \in \mathbf{T}_\Sigma(v, u)$  and  $t' = (t'_1, \dots, t'_p) \in \mathbf{T}_\Sigma(w, v)$ , *composition* is substitution by components:  $t \bullet t' = t''$  where for each  $i \in [n]$

$$t''_i = \{ \langle a, \sigma \rangle \mid \langle a, \sigma \rangle \in t_i \text{ and } \sigma \in \Sigma \} \\ \cup \bigcup_{j \in [p]} \{ \langle ab, \xi \rangle \mid \langle a, x_j^v \rangle \in t_i \text{ and } \langle b, \xi \rangle \in t'_j \}.$$

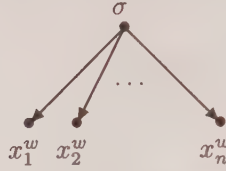
(2.4.5.3) For each  $w \in S^n$  and  $i \in [n]$ , the *distinguished morphism*  $x_i^w : w \rightarrow w_i$  is just the variable  $x_i^w$  which we identify with the  $w$ -ary tree  $\{ \langle \lambda, x_i^w \rangle \}$ .

(2.4.5.4) Tupling is just tupling.

We want to give a category-theoretic characterization of the freeness of  $\mathbf{T}_\Sigma$ . First a little notation. Given any  $S$ -sorted algebraic theory  $\mathbf{T}$ , it is easy to see that the family  $\langle \mathbf{T}(w, s) \mid w \in S^*, s \in S \rangle$  is an  $S$ -sorted signature. Furthermore, if  $F : \mathbf{T}_1 \rightarrow \mathbf{T}_2$  is a theory morphism then the family of mappings  $\langle F_{w,s} : \mathbf{T}_1(w, s) \rightarrow \mathbf{T}_2(w, s) \mid w \in S^*, s \in S \rangle$  is a signature morphism. This, of course, specifies a functor  $U : \mathbf{Th}_S \rightarrow \mathbf{Sig}_S$ . Given an  $S$ -sorted signature  $\Sigma$ , a signature morphism  $f : \Sigma \rightarrow \langle \mathbf{T}(w, s) \mid w \in S^*, s \in S \rangle$  is sometimes written as  $f : \Sigma \rightarrow \mathbf{T}$ . In particular we write



$\eta_\Sigma : \Sigma \rightarrow \mathbf{T}_\Sigma$  for the signature morphism that takes  $\sigma \in \Sigma_{w,s}$  to the tree



**Proposition 2.4.6.** *Let  $\Sigma$  be an  $S$ -sorted signature, then for any  $S$ -sorted Lawvere theory  $\mathbf{T}$ , and any  $(S^* \times S)$ -indexed family of mappings  $f = \langle f_{u,s} : \Sigma_{u,s} \rightarrow \mathbf{T}(u,s) \mid u \in S^*, s \in S \rangle$  there is a unique theory morphism  $f^\sharp : \mathbf{T}_\Sigma \rightarrow \mathbf{T}$  that extends  $f$ . That is, for any  $\sigma \in \Sigma_{u,s}$ , where  $u$  is of length  $|u| = n$ , that  $f^\sharp(\sigma(x_1^u, \dots, x_n^u)) = f(\sigma)$ . Or, using the morphism  $\eta_\Sigma$  and functor  $U$  defined above,  $U(f^\sharp) \bullet \eta_\Sigma = f$ .*

**Example 2.4.7.** Let  $S$  be a set (of sorts) and let  $A = \langle A_s \mid s \in S \rangle$  be an  $S$ -indexed family of sets. For any  $w = w_1 \dots w_n \in S^*$  let  $A^w$  denote the categorical product  $A_{w_1} \times \dots \times A_{w_n}$  in the category of sets and partial functions. (Recall that for sets  $A$  and  $B$ , their product in the category of sets and partial functions has product object is  $A \times B = A \cup (A \otimes B) \cup B$  where  $A \otimes B$  is their cartesian product. The projection  $p_1 : A \times B \rightarrow A$  is then such that  $p_1(\bar{a})$  equals  $\bar{a}$  for  $\bar{a} \in A$ , equals  $a$  for  $\bar{a} = \langle a, b \rangle \in A \otimes B$ , and is undefined on  $B$ , and the projection  $p_2 : A \times B \rightarrow B$  is then such that  $p_2(\bar{a})$  is undefined on  $A$ , equals  $b$  for  $\bar{a} = \langle a, b \rangle \in A \otimes B$ , and equals  $\bar{a}$  for  $\bar{a} \in B$ .) We define  $\mathbf{Pow}_A$  to be the  $S$ -sorted algebraic theory where for each  $u, v \in S^*$ ,  $\mathbf{Pow}_A(u, v)$  is the set of all partial functions  $f : A^u \rightarrow A^v$ . Composition is functional composition, i.e., if  $f : u \rightarrow v$  and  $g : v \rightarrow w$ , then their composite  $g \bullet f$  in  $\mathbf{Pow}_A$ , is the mapping  $gf : A^u \rightarrow A^w$ . Given  $u = u_1 \dots u_n \in S^u$ ,  $v \in S^*$  and  $f_i : v \rightarrow u_i$ ,  $i = 1, \dots, n$ , then,  $(f_1, \dots, f_n)$ , the tupling of  $f_1, \dots, f_n$ , is the mapping

$$(f_1, \dots, f_n) : A^v \rightarrow A^u \\ \bar{a} \in A^v \mapsto \langle f_i(\bar{a}) \mid f_i(\bar{a}) \text{ is defined} \rangle.$$

Finally, for each  $u \in S^u$ , and  $i \in [n]$ , define  $x_i^u : u \rightarrow u_i$  to be the projection map  $x_i^u : A^u \rightarrow A_{u_i}$ .

**Example 2.4.8.** Let  $\Gamma$  be the alphabet

$$\Gamma = \{0, \text{succ}, +, *, \text{if}, \text{then}, \text{else}, \\ \text{true}, \text{false}, \text{is\_zero}, \neg, \wedge, \text{while}, \text{do}, :=, \dot{\cdot}, (, )\},$$

and let  $I$  be a set (of identifiers) disjoint from  $\Gamma$ , and let  $A$  be the set of all subsets of  $(\Gamma \cup I)^*$ . Then, identifying  $A$  with the 1-indexed family of sets  $\langle A_1 = A \rangle$ , we get a corresponding 1-sorted algebraic theory  $\mathbf{Pow}_A$ . Let  $\mathbf{CF}_G$  be the subtheory of  $\mathbf{Pow}_A$  generated by the following mappings:

$$| : A \times A \rightarrow A, \text{ where } |(P, Q) = P \cup Q.$$

$zero : A^\lambda \rightarrow A$ , where  $zero = \{0\}$ .  
 $succ : A \rightarrow A$ , where  $succ(P) = \{\mathbf{succ}(p) \mid p \in P\}$ .  
 $add : A \times A \rightarrow A$ , where  $add(P, Q) = \{(p + q) \mid p \in P, q \in Q\}$ .  
 $mult : A \times A \rightarrow A$ , where  $mult(P, Q) = \{(p * q) \mid p \in P, q \in Q\}$ .  
 $ife : A \times A \times A \rightarrow A$ , where  $ife(P, Q, R)$   
 $= \{(\mathbf{if } p \mathbf{ then } q \mathbf{ else } r) \mid p \in P, q \in Q, r \in R\}$ .  
 $true : A^\lambda \rightarrow A$ , where  $true = \mathbf{true}$ .  
 $false : A^\lambda \rightarrow A$ , where  $false = \mathbf{false}$ .  
 $isz : A \rightarrow A$ , where  $isz(P) = \{\mathbf{is\_zero}(p) \mid p \in P\}$ .  
 $not : A \rightarrow A$ , where  $not(P) = \{\neg(p) \mid p \in P\}$ .  
 $and : A \times A \rightarrow A$ , where  $and(P, Q) = \{(p \wedge q) \mid p \in P, q \in Q\}$ .  
 $id : A \times A \rightarrow A$ , where  $id(P, Q) = \{(p := q) \mid p \in P, q \in Q\}$ .  
 $whl : A \times A \rightarrow A$ , where  $whl(P, Q)$   
 $= \{(\mathbf{while } p \mathbf{ do } q) \mid p \in P, q \in Q\}$ .  
 $com : A \times A \rightarrow A$ , where  $com(P, Q) = \{p, q \mid p \in P, q \in Q\}$ .

We will use this theory in Section 4 to illustrate how the context-free grammar of Example 1.2.3 can be represented by a morphism in an algebraic theory and then solved in the same theory.

## 2.5 Notations and basic identities

From tupling of morphisms one defines the more manageable operation of *pairing*. (c.f. Elgot [?]).

For  $u \in S^n, v \in S^p, w \in S^*$ ,  $\alpha : w \rightarrow u$  and  $\beta : w \rightarrow v$ ,

$$(\alpha, \beta) = (x_1^u \bullet \alpha, \dots, x_n^u \bullet \alpha, x_1^v \bullet \beta, \dots, x_p^v \bullet \beta).$$

And there are corresponding generalizations of the distinguished morphisms which are ‘product projections’.

$$x_{(1)}^{u+v} = (x_1^{u+v}, \dots, x_n^{u+v})$$

$$x_{(2)}^{u+v} = (x_{n+1}^{u+v}, \dots, x_{n+p}^{u+v}).$$

Note the parenthesis around the subscript on  $x_{(i)}^{u+v}$  – this is all that distinguishes  $x_{(1)}^{u+v}$  from  $x_1^{u+v}$ .

It will be convenient later on to push the notation somewhat further and, for example, write

$$x_{(1)}^{u+v+w} : u + v + w \rightarrow u$$

$$x_{(2)}^{u+v+w} : u + v + w \rightarrow v$$

and

$$x_{(3)}^{u+v+w} : u + v + w \rightarrow w$$

for the projections from  $u + v + w$ . Finally, we find it convenient to write

$$x_{(1,3)}^{u+v+w} \text{ for } (x_{(1)}^{u+v+w}, x_{(3)}^{u+v+w})$$

and

$$x_{(2,1,3)}^{u+v+w} \text{ for } (x_{(2)}^{u+v+w}, x_{(1)}^{u+v+w}, x_{(3)}^{u+v+w})$$

Now we collect together some important tupling identities that we will need for proofs to follow. These identities are to be found in Elgot [?]; they have simple proofs from the definitions and the other identities.

**Fact 2.5.1.** Let  $u \in S^n$ ,  $u = u_1 \cdots u_n$ , let  $v, w, t \in S^*$ , and let  $\alpha : w \rightarrow u$ ,  $\beta : w \rightarrow v$ ,  $\gamma : w \rightarrow t$ , and  $\tau : t \rightarrow w$  be morphisms in an algebraic theory  $\mathbf{T}$ . Then,

$$(2.5.1.1) \quad x_{(1)}^{u+v} \bullet (\alpha, \beta) = \alpha \text{ and } x_{(2)}^{u+v} \bullet (\alpha, \beta) = \beta.$$

$$(2.5.1.2) \quad ((\alpha, \beta), \gamma) = (\alpha, (\beta, \gamma))$$

$$(2.5.1.3) \quad (0_w, \alpha) = \alpha = (\alpha, 0_w)$$

$$(2.5.1.4) \quad (\alpha, \beta) \bullet \tau = (\alpha \bullet \tau, \beta \bullet \tau)$$

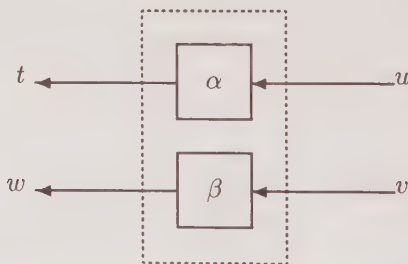
$$(2.5.1.5) \quad 0_u \bullet \alpha = 0_w$$

$$(2.5.1.6) \quad 1_{u+v} = (x_{(1)}^{u+v}, x_{(2)}^{u+v}).$$

The other important operation within algebraic theories is the *product* of morphisms, for:  $\alpha : u \rightarrow t$  and  $\beta : v \rightarrow w$ , their product  $\alpha \times \beta : u+v \rightarrow t+w$  is defined using pairing and projections:

$$(2.5) \quad \alpha \times \beta = (\alpha \bullet x_{(1)}^{u+v}, \beta \bullet x_{(2)}^{u+v}).$$

To picture  $\alpha \times \beta$  it is best to separate the inputs,  $u+v$  into two separate lines, then the picture is



Here are some useful identities involving  $\times$ . We leave the proofs to the reader as a simple exercise.

**Fact 2.5.2.** For  $i \in [3]$  and  $j \in [2]$ , let  $u, v, u_i, v_i, w_j$  and  $w \in S^*$ , and let  $\alpha : v \rightarrow u$ ,  $\alpha_i : v_i \rightarrow u_i$ ,  $\beta_j : w \rightarrow v_j$  and  $\gamma_j : w_j \rightarrow v_j$  be morphisms in an algebraic theory  $\mathbf{T}$ .

$$(2.5.2.1) \quad x_{(1)}^{u+v} = 1_u \times 0_v \text{ and } x_{(2)}^{u+v} = 0_u \times 1_v.$$

$$(2.5.2.2) \quad 0_u \times 0_v = 0_{u+v}.$$

$$(2.5.2.3) \quad 1_u \times 1_v = 1_{u+v}.$$

$$(2.5.2.4) \quad \alpha_1 \times (\alpha_2 \times \alpha_3) = (\alpha_1 \times \alpha_2) \times \alpha_3.$$

$$(2.5.2.5) \quad (\alpha_1 \times \alpha_2) \bullet (\beta_1, \beta_2) = (\alpha_1 \bullet \beta_1, \alpha_2 \bullet \beta_2).$$

$$(2.5.2.6) \quad (\alpha_1 \times \alpha_2) \bullet (\gamma_1 \times \gamma_2) = (\alpha_1 \bullet \gamma_1) \times (\alpha_2 \bullet \gamma_2).$$

$$(2.5.2.7) \quad \alpha \times 1_\lambda = \alpha = 1_\lambda \times \alpha.$$

### 3 Ordered theories

#### 3.1 Definition of ordered and continuous theories

Ordered algebraic theories are just like algebraic theories except that their morphism sets are ordered (are posets) and their operations are monotonic in expected ways. There may be one surprise in that composition is ‘left-strict’: composition by the minimum element on the left gives the minimum element. (A totally undefined computation preceded by anything is still totally undefined.)

##### Definition 3.1.1.

An *ordered algebraic theory* is an algebraic theory where each  $\mathbf{T}(u, v)$  is a strict poset with order relation  $\sqsubseteq_{u,v}$ , and minimum element,  $\perp_{u,v}$ , where composition is left-strict and both composition and target tupling are monotonic.

(3.1.1.1) For all  $u, v, w \in S^*$  and  $\alpha : u \rightarrow v$ ,  $\perp_{v,w} \bullet \alpha = \perp_{u,w}$ .

(3.1.1.2) For all  $u, v, w \in S^*$ ,  $i = 1, 2$ ,  $\alpha_i : u \rightarrow v$ , and  $\beta_i : v \rightarrow w$ ,  
 $\alpha_1 \sqsubseteq \alpha_2$  and  $\beta_1 \sqsubseteq \beta_2$ ,

$$\beta_1 \bullet \alpha_1 \sqsubseteq \beta_2 \bullet \alpha_2.$$

(3.1.1.3) For all  $v = v_1 \cdots v_n \in S^n$ , and  $u \in S^*$   $\alpha_i : u \rightarrow v_i$  and  $\beta_i : u \rightarrow v_i$  with  $i \in [n]$  and  $\alpha_i \sqsubseteq \beta_i$ ,

$$(\alpha_1, \dots, \alpha_n)_{u,v} \sqsubseteq (\beta_1, \dots, \beta_n)_{u,v}$$

Let  $P = \langle P, \sqsubseteq \rangle$  be a poset. An  $\omega$ -chain in  $P$  is an  $\omega$ -indexed family  $\langle p_i \mid i \in \omega \rangle$  of elements of  $P$  such that for all  $i \leq j \in \omega$ ,  $p_i \sqsubseteq p_j$ .  $P$  is  $\omega$ -complete iff every  $\omega$ -chain  $\langle p_i \rangle_{i \in \omega}$  in  $P$  has a least upper bound,  $\bigsqcup_{i \in \omega} p_i$ , in  $P$ . An ordered theory  $\mathbf{T}$  is  $\omega$ -continuous if and only if each  $\mathbf{T}(u, v)$  is  $\omega$ -complete and composition is  $\omega$ -continuous, i.e.,

(3.1.1.4)  $(\bigsqcup_{i \in \omega} \alpha_i) \bullet (\bigsqcup_{i \in \omega} \beta_i) = \bigsqcup_{i \in \omega} (\alpha_i \bullet \beta_i)$  for all  $u, v, q \in S^*$ , and for all  $\omega$ -chains  $\langle \alpha_i \rangle_{i \in \omega}$  and  $\langle \beta_i \rangle_{i \in \omega}$  in  $\mathbf{T}(v, u)$  and  $\mathbf{T}(w, v)$  respectively.

**Definition 3.1.2.** A *morphism*  $F : \mathbf{T}_1 \rightarrow \mathbf{T}_2$  between  $S$ -sorted ordered algebraic theories is a morphism between the two algebraic theories which is strict and monotonic.

(3.1.2.1) Strict: for all  $u, v \in S^*$ ,  $F_{u,v}(\perp_{u,v}) = \perp_{u,v}$ .

(3.1.2.2) Monotonic: for all  $u, v \in S^*$  and  $\alpha, \beta : u \rightarrow v$ , if  $\alpha \sqsubseteq \beta$  then  $F_{u,v}(\alpha) \sqsubseteq F_{u,v}(\beta)$ .

$F$  is a morphism between  $\omega$ -continuous theories iff  $F$  is  $\omega$ -continuous: for all  $u, v \in S^*$  and  $\omega$ -chains  $\langle \alpha_i \rangle_{i \in \omega}$  in  $\mathbf{T}(u, v)$ ,

$$F_{u,v}(\bigsqcup_{i \in \omega} \alpha_i) = \bigsqcup_{i \in \omega} (F_{u,v}(\alpha_i)).$$

$\mathbf{Oth}_S$  denotes the category of  $S$ -sorted ordered algebraic theories with their morphisms and  $\mathbf{Cth}_S$  denotes the category of  $S$ -sorted  $\omega$ -continuous ordered algebraic theories.

While we restrict our attention in this paper to  $\omega$ -continuous theories, most, if not all, of what is done here generalizes to  $Z$ -continuous theories for various  $Z$  as given in [Thatcher *et al.*, 1978].

It is an immediate consequence of Definition 3.1.1 that, in an  $\omega$ -continuous theory, tupling is  $\omega$ -continuous: for all  $u, v \in S^*$ ,  $u = u_1 \cdots u_n$ , and for all  $\omega$ -chains  $\langle \beta_{j,i} \rangle_{i \in \omega}$  for  $j \in [n]$ ,

$$\bigsqcup_{i \in \omega} (\beta_{1,i}, \dots, \beta_{n,i}) = ((\bigsqcup_{i \in \omega} \beta_{1,i}), \dots, (\bigsqcup_{i \in \omega} \beta_{n,i})).$$

**Proposition 3.1.3.** *For each set  $S$  let  $U_{\text{ord}} : \mathbf{OTh}_S \rightarrow \mathbf{Th}_S$  which makes an ordered theory into an ordinary theory by forgetting the ordering. Then  $U_{\text{ord}}$  has a left-adjoint  $F_{\text{ord}} : \mathbf{Th}_S \rightarrow \mathbf{OTh}_S$ . That is, for each  $\mathbf{T} \in |\mathbf{Th}_S|$  there is a morphism  $\eta_{\mathbf{T}} : \mathbf{T} \rightarrow U_{\text{ord}}(F_{\text{ord}}(\mathbf{T}))$  such that for every ordered theory  $\mathbf{T}'$  and ordinary morphism between theories,  $H : \mathbf{T} \rightarrow U_{\text{ord}}(\mathbf{T}')$ , there is a unique morphism  $H^\# : F_{\text{ord}}(\mathbf{T}) \rightarrow \mathbf{T}'$ , in  $\mathbf{OTh}_S$ , such that  $U_{\text{ord}}(H^\#) \bullet \eta_{\mathbf{T}} = H$ .*

**Proof.** [Sketch] The essential idea of the construction is that for all  $u, v \in S^*$  we freely append a new element  $\perp_{u,v}$  to  $\mathbf{T}(u, v)$  and require that  $\perp_{u,v} \sqsubseteq \alpha$  and  $\perp_{u,v} = (\perp_{u,v_1}, \dots, \perp_{u,v_{|v|}})$ . ■

### 3.2 Examples of ordered theories

Several of the algebraic theories given as examples in the preceding section can be viewed as ordered theories. Let  $\sqsubseteq$  be the usual ordering on partial functions, i.e., for  $f, g : X \rightarrow Y$ ,  $f \sqsubseteq g$  iff for each  $x \in X$ , either  $f(x)$  is undefined, or  $f(x) = g(x)$ . Imposing this ordering  $\sqsubseteq$  on the morphisms makes the theories  $\mathbf{Pow}_A$ , and  $\mathbf{Sum}_A$  into ordered theories. Similarly, the theory  $\mathbf{CF}_G$  is an ordered theory with respect to the ordering  $\sqsubseteq$ , where for  $f, g : X \rightarrow Y$ ,  $f \sqsubseteq g$  iff for all  $P \in X$ ,  $f(P) \subseteq g(P)$ .

**Example 3.2.1.** The theory of total, finite  $\Sigma$ -labelled trees is not an ordered theory, but we can define a theory of infinite, partial  $\Sigma$ -labelled trees which is an ordered theory. This theory,  $\mathbf{CT}_\Sigma$  will play an important role in our development.

Let an  $S$ -sorted signature be defined as in Definition 2.4.3. Then the desired trees are defined as follows:

#### Definition 3.2.2.

Let  $\Sigma$  be an  $S$ -sorted signature. For each  $n \in [\omega]$  and  $u \in S^n$  let  $X_u = \{x_1^u, \dots, x_n^u\}$  be an  $[n]$ -indexed set (of variables) disjoint from  $\Sigma_{w,s}$



for all  $w \in S^*$  and  $s \in S$ , and disjoint from all other  $X_v$  for  $v \neq u$ . Call  $x_i^u \in X_u$  the  $i$ th variable of  $X_u$ . Define  $\Sigma \cup X_u$  to be the  $S$ -sorted signature with  $(\Sigma \cup X_u)_{\lambda,s} = \Sigma_{\lambda,s} \cup \{x_i^u \mid u_i = s\}$ , and with  $(\Sigma \cup X_u)_{v,s} = \Sigma_{v,s}$  for  $v \neq \lambda$ .

For each  $w \in S^*$  define the set  $CT_\Sigma(X_w)$  of *partial  $w$ -ary  $S$ -sorted  $\Sigma$ -trees* as the set of all partial functions  $t : [\omega]^* \rightarrow \Sigma \cup X_w$  such that:

for all  $v \in [\omega]^*$ , and  $j \in [\omega]$ , if  $t(vj) = \xi \in (\Sigma \cup X_w)_{v,s}$ , then there exists  $n \geq j$ , and  $u \in S^n$ ,  $q \in S$ , and  $\alpha \in (\Sigma \cup X_w)_{u,q}$ , such that  $u_j = s$  and  $t(v) = \alpha$ .

A partial  $w$ -ary tree  $t$  will be said to be of *sort  $s$*  if either it is nowhere defined, or  $t(\lambda) \in \Sigma_{u,s} \cup \{x_i^u \mid u_i = s\}$  for some  $u \in S^*$ .

**Definition 3.2.3.** With  $S$  a set and  $\Sigma$  an  $S$ -sorted signature,  $\mathbf{CT}_\Sigma$ , the *free  $\omega$ -continuous  $S$ -sorted  $\Sigma$ -(generated-)theory* is defined as follows:

**(3.2.3.1)** For each  $v \in S^n$ ,  $w \in S^*$ ,  $\mathbf{CT}_\Sigma(w, v)$  is the set of all  $n$ -tuples  $(t_1, \dots, t_n)$  of partial  $w$ -ary  $\Sigma$ -trees such that  $t_i$  is of sort  $v_i$  for each  $i \in [n]$ . Let  $\sqsubseteq_{v,w}$  be the ordering on  $\mathbf{CT}_\Sigma(w, v)$  induced by the component-wise ordering on the graphs of partial functions. The *bottom element* is  $\perp_{v,w} = (\perp, \dots, \perp)$  where  $\perp$  is the empty partial function. The order-theoretic properties of  $\mathbf{CT}_\Sigma(w, v)$  are inherited from those of partial functions with the observation that if  $\langle f_i \mid i \in \omega \rangle$  is a chain of partial functions satisfying Definition 3.2.2, then  $\bigsqcup_{i \in \omega} f_i$  also satisfies Definition 3.2.2.

**(3.2.3.2)** For  $u \in S^n$ ,  $v \in S^p$  and  $w \in S^*$ , and for  $t = (t_1, \dots, t_n) \in \mathbf{CT}_\Sigma(v, u)$  and  $t' = (t'_1, \dots, t'_p) \in \mathbf{CT}_\Sigma(w, v)$ , *composition* is substitution by components:  $t \bullet t' = t''$  where for each  $i \in [n]$

$$t''_i = \{ \langle a, \sigma \rangle \mid \langle a, \sigma \rangle \in t_i \text{ and } \sigma \in \Sigma \} \\ \cup \bigcup_{j \in [p]} \{ \langle ab, \xi \rangle \mid \langle a, x_j^v \rangle \in t_i \text{ and } \langle b, \xi \rangle \in t'_j \}.$$

**(3.2.3.3)** For each  $w \in S^n$  and  $i \in [n]$ , the *distinguished morphism*  $x_i^w : w \rightarrow w_i$  is just the variable  $x_i^w$  which we identify with the  $w$ -ary tree  $\{ \langle \lambda, x_i^w \rangle \}$ .

**(3.2.3.4)** Tupling is just tupling.

From  $\mathbf{CT}_\Sigma$  we extract a number of interesting and useful subtheories. A tree  $t \in CT_\Sigma(X_u)$  will be said to be *total* if, it is non-empty and, for all  $v \in [\omega]^*$ , if  $t(v) = \alpha \in \Sigma_{s,u}$ , where  $u \in S^n$ , then for each  $i \in [n]$ ,  $t(vi)$  is defined.

A tree  $t \in CT_\Sigma(X_u)$  will be said to be *finite* if it is only defined on a finite subset of  $[\omega]^*$ .

**Definition 3.2.4.** There are three important subtheories of  $\mathbf{CT}_\Sigma$  that are defined by restricting the class of  $\Sigma$ -trees allowed as morphisms.

**FT $_{\Sigma}$** , where we restrict the morphisms tuples of finite  $\Sigma$ -trees, where a *finite w-ary  $\Sigma$ -tree* is a partial mapping  $t : [\omega]^* \rightarrow \Sigma \cup X_w$  which is defined on only a finite subset of  $[\omega]^*$ . The theory **FT $_{\Sigma}$**  is an ordered theory, indeed it is an ordered theory freely generated by  $\Sigma$ .

**TT $_{\Sigma}$** , where we restrict the morphisms to being tuples of total  $\Sigma$ -trees, where a *total w-ary  $\Sigma$ -tree* is a partial mapping  $t : [\omega]^* \rightarrow \Sigma \cup X_w$  such that for all  $v \in [\omega]^*$  if  $t(v) = \alpha \in \Sigma_{u,s}$  where  $u \in S^n$ , then for each  $i \in [n]$ ,  $t(ui)$  is defined. The theory **TT $_{\Sigma}$**  is not ordered but it contains infinite trees.

**T $_{\Sigma}$** , where we restrict the morphisms to being to being tuples of finite, total trees, defined on a non-empty subset of  $[\omega]^*$ . Note that this is the same algebraic theory as defined in Definition 2.4.5.

All of these theories will play a role in our development.

**Example 3.2.5.** This example provides the appropriate theory in which to formalize the idea of a set of recursion equations and the ‘unwinding’ of a set of recursion equations. At the end of this example we present the recursion equations of Example 1.2.2 as a morphism in this theory. How to ‘unwind’ this morphism to get the desired tree is shown in Subsection 4.4.

It will be convenient to have a special notation for strings of natural numbers: Given a specific string  $v \in \omega^*$  we will write it as  $v_1 \cdot v_2 \cdots v_{|v|}$ , e.g.,  $v = 2 \cdot 1 \cdot 12 \cdot 7$  means that  $v$  is a string in  $\omega^*$ , of length  $|v| = 4$ , and with  $v_1 = 2$ ,  $v_2 = 1$ ,  $v_3 = 12$ , and  $v_4 = 7$ .

**Definition 3.2.6.** Let  $\Sigma = \langle \Sigma_n \mid n \in \omega \rangle$  be a 1-sorted signature. Given  $u \in \omega^*$ , define  $F^u$  to be the 1-sorted signature where, for each  $n \in \omega$ ,  $F_n^u = \{f_i^u \mid i \in [|u|], u_i = n\}$ . Think of  $F^u$  as a signature of *function-variables*. Let  $(\Sigma + F^u)$  be the 1-sorted signature with  $(\Sigma + F^u)_n = \Sigma_n + F_n^u$  for all  $n \in \omega$ . Finally, for each  $p \in \omega$  let  $X_p = \{x_1^p, \dots, x_p^p\}$  be a set of (ground-)variables, as in Definition 3.2.2.

We say a partial tree  $t \in CT_{(\Sigma + F^w)}(X_p)$  is of *rec-sort*  $p$ , and *rec-arity*  $w$ . This says, among other things, that for each  $v \in \omega^*$ , if  $t(v)$  is defined, then either  $t(v) \in (\Sigma \cup X_p)$  or  $t(v) = f_i^w$  for some  $i \in [|w|]$ .

**Definition 3.2.7.** Given  $\Sigma$ ,  $F$ , and  $X$  as above, define **Rec $_{\Sigma}$** , the *free  $\omega$ -continuous 1-sorted  $\Sigma$ -recursion theory* as follows:

- For each  $v \in \omega^n$ ,  $w \in \omega^*$ , **Rec $_{\Sigma}(w, v)$**  is the set of all  $n$ -tuples  $(t_1, \dots, t_n)$  of partial  $(\Sigma + F^w)$ -trees such that, for each  $i \in [n]$ ,  $t_i$  is of rec-sort  $v_i$  and of rec-arity  $w$ .
- For  $u \in \omega^n$ ,  $v \in \omega^p$ , and  $w \in S^*$ , and for  $t' = (t'_1, \dots, t'_p) \in \mathbf{Rec}_{\Sigma}(w, v)$  and  $t = (t_1, \dots, t_n) \in \mathbf{Rec}_{\Sigma}(v, u)$  define their *composition*,  $t \bullet t' = t'' = (t''_1, \dots, t''_n)$ , where, letting  $\circ$  denote the composition operation in **CT $_{\Sigma+F}$** ,

$$t''_i = t_i \bullet t' = \begin{cases} t_i & \text{if } t_i(\lambda) \in \Sigma_0 \cup X, \\ \gamma \circ (\tau_1 \bullet t', \dots, \tau_q \bullet t') & \text{if } t_i = \gamma \circ (\tau_1, \dots, \tau_q), \\ & \gamma \in \Sigma_q \\ t'_j \circ (\tau_1 \bullet t', \dots, \tau_q \bullet t') & \text{if } t_i = f_j^v \circ (\tau_1, \dots, \tau_q). \end{cases}$$

The idea is that we are substituting for function-variables (in the middle of the trees) rather than for ground-variables at the leaves of the trees. Or, to put it another way,  $t_i \bullet t'$  is the result of simultaneously replacing, for all  $j$ , the occurrences of  $f_j^v$  in  $t_i$  by  $t'_j$ .

- Tupling is, again, just tupling.
- Finally, for  $u = u_1 \cdots u_n \in \omega^n$ , the  $i$ th distinguished morphism, which we denote as  $f_i^u : u \rightarrow u_i$ , is

$$f_i^u = f_i^u \circ (x_1^n, \dots, x_n^n).$$

Note that we have not been quite precise in our notation, we can not always determine the rec-arity or rec-sort of a morphism by inspection, and so we should define  $\mathbf{Rec}_\Sigma(u, w)$  as a set of triples  $\langle u, (t_1, \dots, t_n), v \rangle$  so that the source and target information is explicit. However, the ambiguous notation is less cumbersome.

To make this a bit more concrete consider the case where  $\Sigma$  is the one-sorted signature with

$$\begin{aligned} \Sigma_0 &= \{0, 1\}, \\ \Sigma_1 &= \{pred, succ\}, \\ \Sigma_4 &= \{if\}. \end{aligned}$$

Then the set of equations

$$\begin{aligned} f_1(x_1^2, x_2^2) &\equiv \text{If } x_1^2 = 0 \text{ then } x_2^2 \text{ else } succ(f_1(pred(x_1^2), x_2^2)), \\ f_2(x_1^2, x_2^2) &\equiv \text{If } x_1^2 = 0 \text{ then } 0 \text{ else } f_1(f_2(pred(x_1^2), x_2^2), x_2^2), \\ f_3(x_1^1) &\equiv \text{If } x_1^1 = 0 \text{ then } 1 \text{ else } f_2(x_1^1, f_3(pred(x_1^1))). \end{aligned}$$

from Example 1.2.2, is represented by a morphism

$$\alpha = (\alpha_1, \alpha_2, \alpha_3) : 2 \cdot 2 \cdot 1 \rightarrow 2 \cdot 2 \cdot 1$$

where, letting  $u = 2 \cdot 2 \cdot 1$ ,

$$\begin{aligned} \alpha_1 &= if(x_1^2, 0, x_2^2, succ(f_1^u(pred(x_1^2), x_2^2))), \\ \alpha_2 &= if(x_1^2, 0, 0, f_1^u(f_2^u(pred(x_1^2), x_2^2), x_2^2)), \\ \text{and} \\ \alpha_3 &= if(x_1^1, 0, succ(0), f_2^u(x_1^1, f_3^u(pred(x_1^1)))). \end{aligned}$$

To help make things look more familiar, we have denoted composition by juxtaposition rather than by  $\circ$ . That is, for example, following the conventions used in most of this paper, we would write  $\alpha_1$  as:

$$if \circ (x_1^2, 0, x_2^2, succ \circ (f_1^u \circ (pred \circ (x_1^2), x_2^2))).$$

or, deleting the unneeded parentheses, as:

$$if \circ (x_1^2, 0, x_2^2, succ \circ f_1^u \circ (pred \circ x_1^2, x_2^2)).$$

## 4 Theories with iteration operators

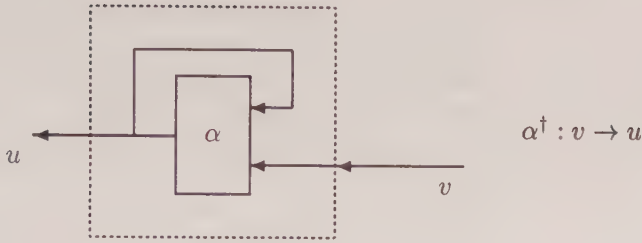
### 4.1 Iteration operators

The algebraic theories we have looked at so far permit us to model such objects as flowcharts without feedback, or, equivalently, straightline programs. We now want to look at the more general situation where feedback is permitted, or, equivalently we allow programs to have loops. The most general treatment is afforded by the concept of an iteration theory developed by Bloom, Elgot and Wright [1980a; 1980b] and axiomatized by Ésik [1980]. Iteration theories axiomatize the concept of an iteration operator ' $\dagger$ '. Given a morphism  $\alpha : u + v \rightarrow u$ , its iterate,  $\alpha^\dagger : v \rightarrow u$ , is a fixpoint for  $\alpha$  in the sense that we always have  $\alpha \bullet (\alpha^\dagger, 1_v) = \alpha^\dagger$ . While the axiomatization is complete in the sense that it provides a basis for proving all the identities that hold for iteration, it does not say anything about 'how the fixpoint is achieved'. However, there are two more specialized concepts which do say something about that issue: first we have the iterative theories of Elgot [Elgot, 1975] where the fixpoints are unique (for appropriate  $\alpha$ ), and second we have the rational theories of ADJ [Wright *et al.*, 1976] where the fixpoints are minimal.

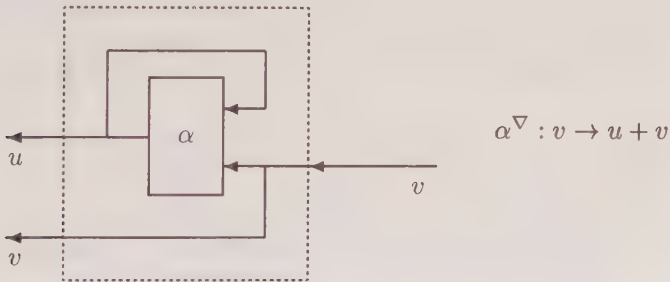
Before introducing any formal definitions let us again look at the 'black box' version of iteration operators. The intuition is as follows: If we have a morphism  $\alpha : u + v \rightarrow u$  in an algebraic theory  $\mathbf{T}$ , and we picture  $\alpha$  as a 'black box'



then we can form a new 'black box',  $\alpha^\dagger$ , the iterate of  $\alpha$ , by 'connecting the  $u$  outputs back to the  $u$  inputs so as to get feedback', as shown below.



A slight, but very useful, modification of  $\alpha^\dagger$  is the morphism  $\alpha^\nabla$ , which results from ‘feeding through the  $v$  inputs’ in addition to ‘feeding back the  $u$  outputs’. This corresponds to the following picture.



Our aim now is to look at various ways in which to define algebraic theories in which  $\alpha^\dagger$  and  $\alpha^\nabla$  exist for all morphisms, or at least exist for some broad class of morphisms. However, we want to do this in a way that reflects our intuitions. For example, if we take  $\alpha$  to be the theory morphism in  $\mathbf{CT}_\Sigma$  as in Example 3.2.3 representing the flowchart given in Example 1.2.1, then it would seem reasonable to take  $\alpha^\dagger$  to be the morphism in  $\mathbf{CT}_\Sigma$  corresponding to the infinite flowchart we get from ‘unwinding’  $\alpha$ .

## 4.2 Iteration, rational, and iterative theories

In this subsection we present three different ways of looking at iteration within algebraic theories. We start with Ésik’s axiomatization [?] of the *iteration theories* introduced in [Bloom *et al.*, 1980a; Bloom *et al.*, 1980b]. Iteration theories provide an equational axiomatization for iteration operators. Second we present rational theories, developed by ADJ [Wright *et al.*, 1976], where ordered theories are used to capture the idea of defining iteration in terms of least fixpoints. Finally we present Elgot’s iterative theories [?] which where the theories are such that the ‘interesting fixpoints’ are unique.



### 4.2.1 Iteration theories

Iteration theories capture the concept of an iteration operator axiomatically, indeed equationally, and thus, like the original Lawvere algebraic theories, they are a special variety of many-sorted algebras.

**Definition 4.2.1.** ([Ésik, 1980]—modified) An *iteration theory* is an algebraic theory equipped with an  $(S^* \times S^*)$ -indexed family of operations

$$\dagger_{u,v} : \mathbf{T}(u + v, u) \rightarrow \mathbf{T}(v, u),$$

called *iteration (operations)*. Given  $\alpha : u + v \rightarrow u$ , we shall, henceforth, write  $\alpha^\dagger$  rather than  $\dagger_{u,v}(\alpha)$ , leaving the subscripts to the context. The iteration operations are subject to the following axioms, which we first state and then motivate:

(4.2.1.1)  $\alpha^\dagger = \alpha \bullet (\alpha^\dagger, 1_v)$  where  $\alpha : u + v \rightarrow u$ .

(4.2.1.2) (The ‘key identity’): If  $\alpha : u + v + w \rightarrow u$  and  $\beta : u + v + w \rightarrow v$  then

$$(\alpha, \beta)^\dagger = (\alpha^\dagger \bullet ((\beta \bullet (\alpha^\dagger, 1_{v+w}))^\dagger, 1_w), (\beta \bullet (\alpha^\dagger, 1_{v+w}))^\dagger).$$

(4.2.1.3)  $(0_u \times \alpha)^\dagger = \alpha$ , for  $\alpha : v \rightarrow u$ .

(4.2.1.4)  $(\alpha \times 0_w)^\dagger = (\alpha^\dagger \times 0_w)$ , for  $\alpha : u + v \rightarrow u$ .

(4.2.1.5) Given  $u, w \in S^*$ ,  $v \in S^m$ ,  $\beta : v + w \rightarrow u$ , and base morphisms  $f : u \rightarrow v$ , and  $g_1, \dots, g_m : v \rightarrow v$  such that  $g_1 \bullet f = \dots = g_m \bullet f = f$ , and  $f$  is surjective (i.e., for every  $i = 1, \dots, |u|$ , there exists  $j$ ,  $1 \leq j \leq |v|$  such that  $x_j^v \bullet f = x_i^u$ ), then

$$(x_1^v \bullet f \bullet \beta \bullet (g_1 \times 1_w), \dots, x_m^v \bullet f \bullet \beta \bullet (g_m \times 1_w))^\dagger = f \bullet (\beta \bullet (f \times 1_w))^\dagger.$$

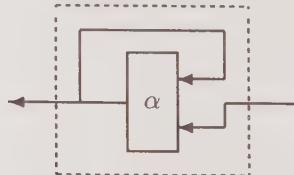
(4.2.1.6)  $(\alpha \bullet (1_u \times \tau))^\dagger = \alpha^\dagger \bullet \tau$ , for all  $\alpha : u + v \rightarrow u$  and  $\tau : w \rightarrow u$ .

A *morphism of  $S$ -sorted iteration theories*  $F : \mathbf{T}_1 \rightarrow \mathbf{T}_2$  is a morphism of algebraic theories from  $\mathbf{T}_1$  to  $\mathbf{T}_2$  which preserves the iteration operator, that is, for all  $\alpha \in \mathbf{T}(u + v, u)$

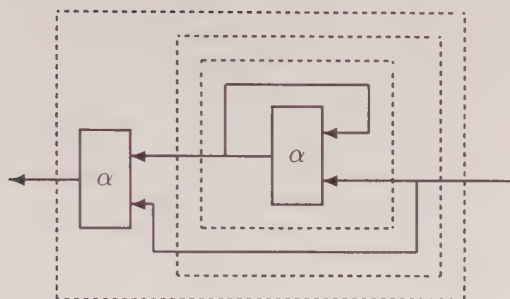
$$F(\alpha^\dagger) = (F(\alpha))^\dagger.$$

Let  $\mathbf{Ith}_S$  denote the category of  $S$ -sorted iteration theories.

The intuitive content of Axiom 4.2.1.1 is that  $\alpha^\dagger$  is a fixpoint for  $\alpha$ . This corresponds to the ‘black box circuits’

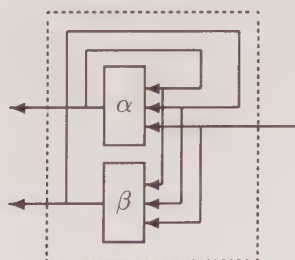


and

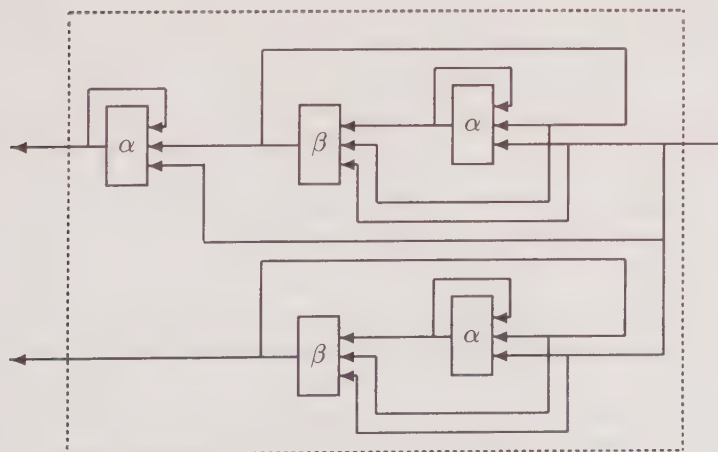


being 'behaviourally equivalent'. To put it another way, the axioms may be thought of as asserting that partially 'unwinding' the circuit (iteration, recursion, etc.) does not change its behaviour.

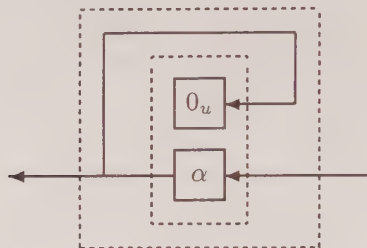
Axiom 4.2.1.2 is again a matter of 'partial unwinding' but it is rather harder to see. Essentially it asserts the equivalence of the circuits



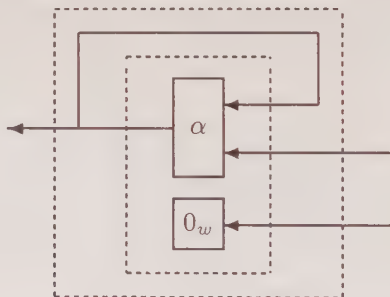
and



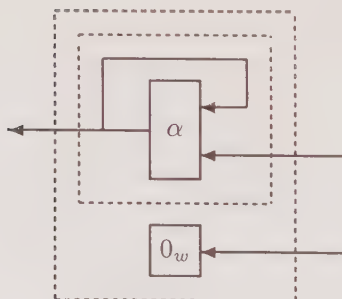
Axioms 4.2.1.3 and 4.2.1.4 are relatively straightforward. The first asserts that the 'circuit'



is equivalent to  $\alpha$ , i.e., that 'feeding back' the output through the 'output-less box'  $0_u$  has no effect. The second is even more 'visually obvious' as it asserts the equivalence of the 'circuits'

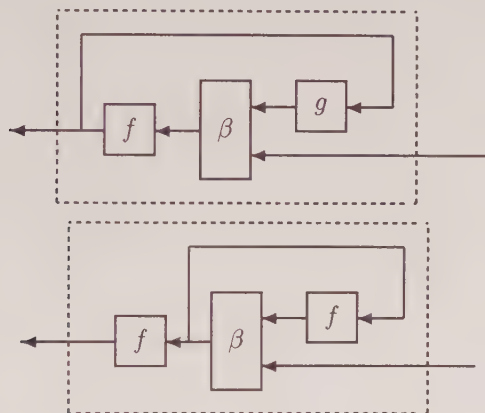


and

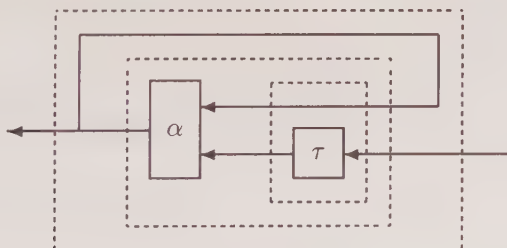


What makes the axiom 'visually obvious' is that the 'circuit' is the same in both pictures and all that differs is the placement of the 'boxes' corresponding to the operations used to construct the 'circuit'. The mathematical content of the axiom is, of course, that it enforces this highly desirable property of the constructions in a general framework that extends far beyond the simple intuitive example provided by the 'circuits'.

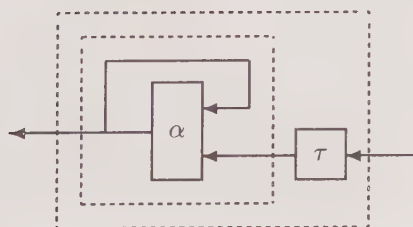
Axiom 4.2.1.5 is a technical axiom, with little intuitive content. The intuition is easiest to see in the case that  $m = 1$ , where it amounts to the following two ‘circuits’ having the same meaning when  $g \bullet f = f$ :



Finally, Axiom 4.2.1.6 is again ‘visually obvious’ in that it asserts the equivalence of the circuits



and



Another intuitive way to view this axiom is as say that, that given  $\alpha : u + v \rightarrow v$ , we get the same result if we ‘fix’ the second argument to  $\tau$  and

then apply  $\dagger$ , as we do when we first apply  $\dagger$  and then apply the result to  $\tau$ .

The above definition of iteration theory differs from that of Ésik [Ésik, 1980] in that we use a different form of Axiom 4.2.1.2 than used by Ésik, and we add Axiom 4.2.1.6. While Axiom 4.2.1.6 can be derived from the other axioms (see [Ésik, 1980]) it simplifies our development to have it as an axiom. For that matter, as Ésik shows in [Ésik, 1983], Axiom 4.2.1.1 is also redundant. For an alternative axiomatization see Ștefănescu, [1987].

With the exception of  $\mathbf{Str}_A^{op}$ , all the algebraic theories in the preceding section can be made into iteration theories by the addition of appropriate iteration operators. We will look at some examples in more detail after defining some other important classes of theories with iteration operators.

#### 4.2.2 Rational theories

One way of forming an iteration theory is to work with special ordered algebraic theories where the iteration operator can be defined in terms of least fixed points. This idea is captured by the concept of a rational algebraic theory.

**Definition 4.2.2.** [Wright *et al.*, 1976] Let  $\mathbf{T}$  be an  $S$ -sorted ordered theory, and let  $\alpha : u + v \rightarrow u$  in  $\mathbf{T}$ . Then by the  $k$ th iterate of  $\alpha$  we mean the morphism

$$\alpha^{(k)} = \alpha \bullet (\alpha, x_{(2)}^{u+v})^k \bullet (\perp_{v,u}, 1_v) : v \rightarrow u.$$

Intuitively,  $\alpha^{(k)}$  is the result of the first  $k$  steps in the ‘unwinding’ of  $\alpha$ . Indeed, if we ‘unwind’ a morphism  $\alpha$  in the free  $\omega$ -continuous theory  $\mathbf{CT}_\Sigma$  then  $\alpha^{(k)}$  is the partial tree that we get from the first  $k$  steps of unwinding  $\alpha$  (see below).

We call  $\langle \alpha^{(k)} \mid k \in \omega \rangle$  the *rational sequence associated with  $\alpha$* .

An ordered theory  $\mathbf{T}$  will be said to be a *rational theory* if for all  $u, v, w \in S^*$ , and  $\alpha : u + v \rightarrow u$ ,  $\eta : v \rightarrow u$ , and  $\tau : w \rightarrow v$ ,

(4.2.2.1)  $\alpha^\dagger = \bigsqcup \alpha^{(k)}$  exists in  $\mathbf{T}$ ;

(4.2.2.2)  $\alpha \bullet (\alpha^\dagger, 1_v) = \alpha^\dagger$ ;

(4.2.2.3) if  $\alpha \bullet (\eta, 1_v) \sqsubseteq \eta$  then  $\alpha^\dagger \sqsubseteq \eta$ ;

(4.2.2.4)  $(\alpha \bullet (1_u \times \tau))^\dagger = \alpha^\dagger \bullet \tau$ .

We call  $\alpha^\dagger$  the *minimal solution for  $\alpha$* .

A morphism between rational theories is an ordered theory morphism (see Definition 3.1.2) which preserves  $\dagger$ . Since, for  $\alpha : u + v \rightarrow u$ ,  $\alpha^\dagger = \bigsqcup \alpha^{(k)}$ ,  $\alpha^{(k)}$  as above, preservation of  $\dagger$  is the same as preservation of least upper bounds of rational sequences. Let  $\mathbf{RTh}_S$  denote the category of  $S$ -sorted rational theories,  $\mathbf{RTh}_S$ .

The intuition behind the axioms for rational theories is fairly simple. Given  $\alpha : u + v \rightarrow u$ , Axiom 4.2.2.1 asserts the existence of a least upper bound for the rational sequence associated with  $\alpha$  and defines  $\alpha^\dagger$  to be



that upper bound. This says then that the rational sequence has a limit in  $\mathbf{T}$ . Axiom 4.2.2.2 says that  $\alpha^\dagger$ , so defined, is a fixpoint for  $\alpha$ , and Axiom 4.2.2.3 says, in effect, that it is the least fixpoint (the actual axiom making a slightly, but significantly, stronger assertion). For the intuition behind Axiom 4.2.2.4 see the discussion of Axiom 4.2.1.6 for iteration theories.

The following lemma provides an alternative definition of  $\alpha^{(k)}$  that has a more recursive flavor.

**Lemma 4.2.3.** *In any ordered theory  $\mathbf{T}$ , if  $\alpha : u + v \rightarrow u$ , then for all  $k \in \omega$ ,*

$$(4.2.3.1) \quad x_{(2)}^{u+v} \bullet (\alpha, x_{(2)}^{u+v})^k \bullet (\perp_{v,u}, 1_v) = 1_v;$$

$$(4.2.3.2) \quad \alpha^{(k+1)} = \alpha \bullet (\alpha^{(k)}, 1_v).$$

**Proof.** The first part follows with induction immediately from (4.2.2) and (2.5.1.1). For the second part, the case  $k = 0$  comes directly from (4.2.2),  $\alpha^{(0)} = \alpha(\perp_{v,u}, 1_v)$  and

$$\begin{aligned} & \alpha^{(1)} \\ &= \alpha \bullet (\alpha, x_{(2)}^{u+v}) \bullet (\perp_{v,u}, 1_v) & (4.2.2) \\ &= \alpha \bullet (\alpha \bullet (\perp_{v,u}, 1_v), x_{(2)}^{u+v} \bullet (\perp_{v,u}, 1_v)) & (2.5.1.4) \\ &= \alpha \bullet (\alpha^{(0)}, 1_v) & (2.5.1.1) \end{aligned}$$

For an inductive proof, assume 4.2.3 for  $k$  and consider  $k + 1$ .

$$\begin{aligned} & \alpha^{(k+1)} \\ &= \alpha \bullet (\alpha, x_{(2)}^{u+v})^{k+1} \bullet (\perp_{v,u}, 1_v) & (4.2.2) \\ &= \alpha \bullet (\alpha, x_{(2)}^{u+v}) \bullet ((\alpha, x_{(2)}^{u+v})^k \bullet (\perp_{v,u}, 1_v)) \\ &= \alpha \bullet (\alpha^{(k)}, 1_v) & (2.5.1.4, 4.2.2, 4.2.3.1) \end{aligned}$$

as required. ■

**Example 4.2.4.** The algebraic theories  $\mathbf{Sum}_A$ ,  $\mathbf{CT}_\Sigma$ , and  $\mathbf{Rec}_\Sigma$  are examples of rational theories.

**Proposition 4.2.5.** *Every  $\omega$ -continuous algebraic theory (see Definition 3.1.1) is a rational algebraic theory.*

We will see, later on, that there are iteration theories that are not rational theories.

### 4.2.3 Iterative theories

We now give the definition of iterative theory as developed by Elgot [Elgot, 1975]. The idea here is to define the iteration operator in terms of fixpoints and to restrict our attention to algebraic theories in which the fixpoints are unique. Unfortunately, every algebraic theory contains certain morphisms which do not have unique fixpoints. For example, the morphism  $x_{(1)}^{u+v} :$

$u + v \rightarrow u$  has every morphism  $\gamma : v \rightarrow u$  as a fixpoint:  $x_{(1)}^{u+v} \bullet (\gamma, 1_v) = \gamma$  by the definition of  $x_{(1)}^{u+v}$ . Thus to employ  $\dagger$  to denote unique fixpoints requires viewing it as a partial operator defined only on certain morphisms. In Elgot's iterative theories the iteration operator is defined on the special class of morphisms that Elgot calls ideal morphisms. The partiality of  $\dagger$  means that iterative theories are not iteration theories, but, as we will see, they are closely related.

**Definition 4.2.6.** ([Elgot, 1975] Let  $u \in S^n$  and  $v \in S^*$ , then a morphism  $\alpha : v \rightarrow u$  in  $\mathbf{T}$  is *ideal* if, for each  $i \in [n]$ , and each  $\beta : w \rightarrow v$ ,  $w \in S^*$ ,  $x_i^u \bullet \alpha \bullet \beta$  is not a distinguished morphism. An algebraic theory  $\mathbf{T}$  is *ideal* if every nondistinguished morphism in  $\mathbf{T}$  is ideal. An ideal theory  $\mathbf{T}$  is an *iterative* theory if for every ideal morphism  $\alpha : u + v \rightarrow u$  in  $\mathbf{T}$  there is a unique morphism  $\alpha^\dagger : v \rightarrow u$  such that

$$\alpha \bullet (\alpha^\dagger, 1_v) = \alpha^\dagger.$$

We call  $\alpha^\dagger$  *the solution for  $\alpha$* .

A *morphism of iterative theories*  $F : \mathbf{T}_1 \rightarrow \mathbf{T}_2$  is a theory morphism  $F$  from  $\mathbf{T}_1$  to  $\mathbf{T}_2$  which preserves iteration applied to ideal morphisms, i.e., if  $\alpha : u + v \rightarrow u$  is ideal, then  $F(\alpha^\dagger) = (F(\alpha))^\dagger$ . Let  $\mathbf{ETh}_S$  denote the **category of  $S$ -sorted iterative theories** (or *Elgot-theories*).

**Example 4.2.7.** The theory  $\mathbf{TT}_\Sigma$  of total infinite trees is an iterative theory.

### 4.3 The $\nabla$ -operator

To reduce the size of expressions, and to increase readability we introduce

$$(4.3.1) \quad \alpha^\nabla = (\alpha^\dagger, 1_v).$$

for  $\alpha : u + v \rightarrow v$ .

For example, Axiom 4.2.1.2, the key identity, can be written as

$$(\alpha, \beta)^\dagger = (\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla, (\beta \bullet \alpha^\nabla)^\dagger).$$

We immediately get

$$(4.3.2) \quad \alpha \bullet \alpha^\nabla = \alpha^\dagger = x_{(1)}^{u+v} \bullet \alpha^\nabla$$

in any iteration, iterative, or rational, theory.

The  $\nabla$ -operator will play a particularly important role in the final part of this section.

### 4.4 Iteration operators and flowcharts

Some informal discussion is needed to sharpen our intuitive understanding of the relationship between ‘flowcharts’ and theory morphisms.

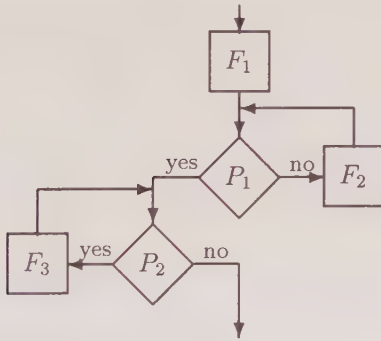
The idea we want to promote is that the proper notion of a ‘flowchart’

in an  $S$ -sorted iteration theory  $\mathbf{T}$  is that captured in the following definition:

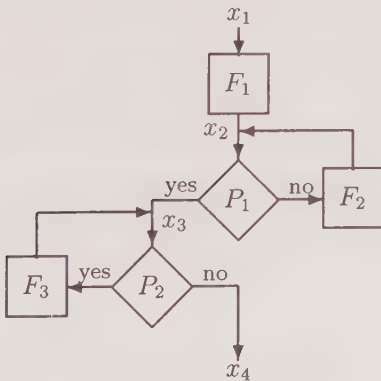
**Definition 4.4.1.** Let  $\mathbf{T}$  be an  $S$ -sorted iteration theory. Define an *abstract flowchart in  $\mathbf{T}$*  with  $w$  inputs and  $v$  outputs as a pair of morphisms  $\langle \beta : u + v \rightarrow w, \alpha : u + v \rightarrow u \rangle$  for some  $u \in S^*$ . Define the *semantics*, in  $\mathbf{T}$ , of the abstract flowchart  $\langle \beta : u + v \rightarrow w, \alpha : u + v \rightarrow u \rangle$  as the morphism  $\beta \bullet \alpha^\nabla$ .

Our claim is that this is the right notion for any choice of  $\mathbf{T}$ , and so, in particular, it is the right way to view ‘ordinary flowcharts’, context free grammars, and recursion equations. Let us look at these examples starting with ‘ordinary flowcharts’.

**Example 4.4.2.** Recall that in the introduction we unwound the flowchart

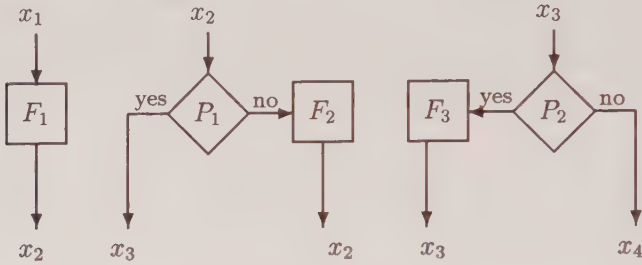


by first marking it so as to ‘cut all the loops’



Then we broke the flowchart into a 4-tuple of trees, including one for the ‘output’  $x_4$ . Here we will take a slightly different approach and will only produce a tree for the input  $x_1$  and for the two loop-cuts  $x_2$  and  $x_3$ , thus

we break the flowchart into a 3-tuple of trees:



This 3-tuple corresponds to a theory morphism

$$\alpha = (F_1(x_2^4), P_1(x_3^4, F_2(x_2^4)), P_2(F_3(x_3^4), x_4^4)) : 4 \rightarrow 3$$

in  $\mathbf{CT}_\Sigma$ , where  $\Sigma$  is one-sorted signature with  $F_1, F_2, F_3 \in \Sigma_{1,1}$  and  $P_1, P_2 \in \Sigma_{2,1}$  (see Definition 3.2.3 for the definition of  $\mathbf{CT}_\Sigma$ ). Note that the direction of the arrow in  $\alpha : 4 \rightarrow 3$  is opposite to that of the arrows in the flowchart. This could be ‘corrected’ by changing to the dual definition of algebraic theory—but then the next example, that of recursion equations, would ‘come out backwards’.

The 4-tuple of trees used in Section 1 is then represented by the  $\mathbf{CT}_\Sigma$ -morphism  $(\alpha, x_4^4) : 4 \rightarrow 4$ .

Now the morphism  $\alpha$ , by itself, does not carry all the information in the original picture because it does not say, explicitly, which of the  $x_i^4$  is the ‘entrance’ to the flowchart. To make the ‘entrance’ explicit we can give an additional morphism  $\beta : 4 \rightarrow 1$ . (Yes, we do mean  $\beta : 4 \rightarrow 1$  rather than  $\beta : 3 \rightarrow 1$ —see below.) In this example we can take  $\beta = x_1^4$ . Thus the complete flowchart is described by the pair  $\langle \beta, \alpha \rangle$ .

Furthermore, we claim that the infinite tree corresponding to the unwinding of the flowchart is precisely the tree corresponding to the expression  $\beta \bullet \alpha^\nabla$ .

To see why this is so, or at least to get some intuition why this is so, it is advantageous to look at  $\mathbf{CT}_\Sigma$  as a rational theory. Then we know that

$$\alpha^\nabla = (\alpha^\dagger, 1_v)$$

where

$$\alpha^\dagger = \bigsqcup \alpha^{(k)}$$

where, in this case,

$$\alpha^{(k)} = \alpha \bullet (\alpha, x_{(2)}^{3+1})^k \bullet (\perp_{1,3}, 1_1)$$

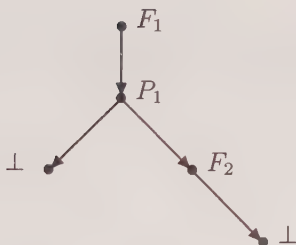
But, because  $\mathbf{CT}_\Sigma$  is  $\omega$ -continuous (Definition 3.1.1) we then have

$$\begin{aligned}
 \beta \bullet \alpha^\nabla &= \beta \bullet (\alpha^\dagger, 1_1) && \text{def } \alpha^\nabla \\
 &= x_1^4 \bullet (\alpha^\dagger, 1_1) && \beta = x_1^4 \\
 &= x_1^4 \bullet (\bigsqcup \alpha^{(k)}, 1_1) && \text{def } \alpha^\dagger \\
 &= x_1^3 \bullet \bigsqcup \alpha^{(k)} && \text{def } x_1^4 \\
 &= \bigsqcup (x_1^3 \bullet \alpha^{(k)}) && (3.1.1.4) \\
 &= \bigsqcup (\alpha_1 \bullet (\alpha, x_4^4)^k \bullet (\perp_{1,3}, 1_1)) && \text{def } \alpha^\dagger
 \end{aligned}$$

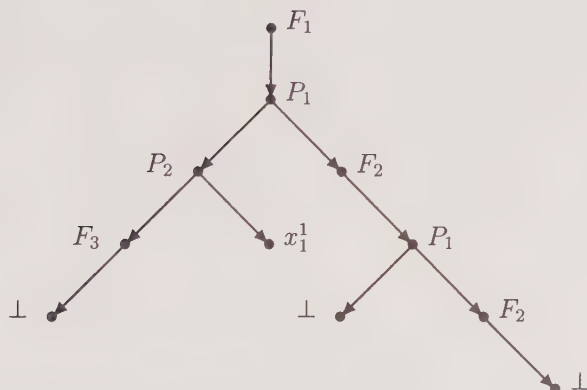
But then, the successive approximates of  $\beta \bullet \alpha^\nabla$  start out as follows:  
for  $k = 0$ :



for  $k=1$ :



for  $k=2$ :



for  $k=3$ :





course, different choices give different results, but they are all, none-the-less, appropriate.

The idea of what  $\beta$  does is easier to see when the theory is interpreted. So, looking ahead, if we interpret  $0$ ,  $\text{succ}$ ,  $\text{pred}$ , and  $\text{if}$  as the usual operations on the set  $\mathbf{N}$  of natural numbers, then  $f_{2,1}$  will be interpreted as addition,  $f_{2,2}$  will be interpreted as multiplication, and  $f_{1,3}$  will be interpreted as factorial. Then, for example, taking, say,  $w = 2 \cdot 1$ , and

$$\beta = (f_{2,1}(\text{succ}(\text{succ}(x_1^2))), f_{1,3}(x_2^2)), f_{2,2}(x_1^1, x_1^1))$$

will give us a flowchart  $\langle \beta, \alpha \rangle$  with meaning  $\beta \bullet \alpha^\nabla : \lambda \rightarrow 2 \cdot 1$ . That is, the interpretation of  $\beta \bullet \alpha^\nabla$  will be a pair of functions  $\langle g_1 : \omega^2 \rightarrow \omega, g_2 : \omega \rightarrow \omega \rangle$  such that for any  $n, p \in \mathbf{N}$ ,  $g_1(n, p) = (n + 2) + (p!)$  and  $g_2(n) = n^2$ .

More generally, a 'flowchart'  $\langle \beta, \alpha \rangle$  in  $\mathbf{Rec}_\Sigma$  can be viewed as a 'set'  $\alpha$  of recursion equations defining a tuple of functions, together with an 'application' of these functions to either define other functions or to compute a value. In the case above we have  $\alpha : u + v \rightarrow u$  with  $v = \lambda$ , the empty string on  $\omega$ . When  $v \neq \lambda$  then  $\alpha$  is a parameterized set of recursion equations, in the sense that the interpretation of  $\beta \bullet \alpha^\nabla : v \rightarrow w$  would be that it is an operation that takes functions on  $\mathbf{N}$  of the arities specified by  $v$  and returns functions of the arities specified by  $w$ .

Now, back to the uninterpreted case, and the solution of  $\alpha$ . In our example we have  $\alpha : u + \lambda \rightarrow u$  with  $u = 1 \cdot 2 \cdot 2$ . Thus

$$\alpha^\nabla = (\alpha^\dagger, 1_\lambda) = (\alpha^\dagger, 0_\lambda) = \alpha^\dagger$$

and, for each  $k \in \omega$ ,

$$\alpha^{(k)} = \alpha \bullet (\alpha, x_{(2)}^{u+\lambda})^k \bullet (\perp_{\lambda, u}, 1_\lambda) = \alpha^{k+1} \bullet (\perp_{\lambda, u}).$$

Which, in turn, yields

$$\alpha^\dagger = \bigsqcup \alpha^{(k)} = \bigsqcup \alpha^k (\perp_{\lambda, u}),$$

which probably looks quite familiar.

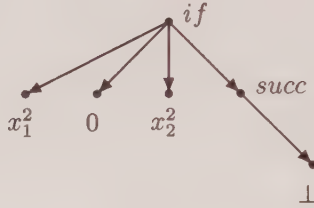
The question then is, what is  $\alpha^k \bullet (\perp_{\lambda, u})$  in  $\mathbf{Rec}_\Sigma$ ? To make the example a bit more tractable, we will look at the simpler example given by

$$\begin{aligned} \alpha &= \text{if}(x_1^2, 0, x_2^2, \text{succ}(f_{2,1}(\text{pred}(x_1^2), x_2^2))) \\ &= \text{if} \circ (x_1^2, 0, x_2^2, \text{succ} \circ f_{2,1} \circ (\text{pred} \circ (x_1^2), x_2^2)), \end{aligned}$$

and we shall drop the subscripts on  $\perp_{\lambda, u}$ . Then, by the definition of  $\bullet$  in  $\mathbf{Rec}_\Sigma$ , Definition 3.2.7, we have

$$\begin{aligned} \alpha^{(0)} &= \alpha^1 \bullet \perp \\ &= \text{if} \circ (x_1^2 \bullet \perp, 0 \bullet \perp, x_2^2 \bullet \perp, \\ &\quad (\text{succ} \circ f_{2,1} \circ (\text{pred} \circ (x_1^2), x_2^2)) \bullet \perp) \\ &= \text{if} \circ (x_1^2, 0, x_2^2, \text{succ} \circ ((f_{2,1} \circ (\text{pred} \circ (x_1^2), x_2^2)) \bullet \perp)) \\ &= \text{if} \circ (x_1^2, 0, x_2^2, \text{succ} \circ \perp \circ ((\text{pred} \circ (x_1^2), x_2^2) \bullet \perp)) \\ &= \text{if} \circ (x_1^2, 0, x_2^2, \text{succ} \circ \perp) \end{aligned}$$

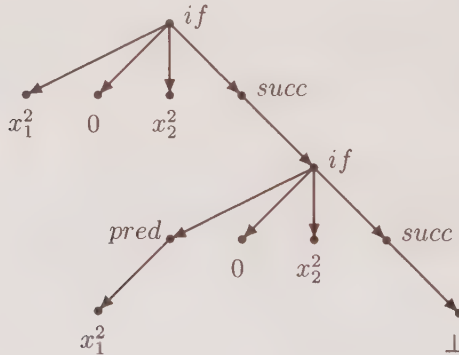
since  $\perp$  is left-strict i.e.,  $\perp \circ \gamma = \perp$ , for all  $\gamma$ . Thus  $\alpha^{(0)}$  corresponds to the tree



Continuing we get

$$\begin{aligned}
 \alpha^{(1)} &= \alpha^2 \bullet \perp \\
 &= \alpha \bullet (\alpha \bullet \perp) \\
 &= if \circ (x_1^2 \bullet (\alpha \bullet \perp), 0 \bullet (\alpha \bullet \perp), x_2^2 \bullet (\alpha \bullet \perp), \\
 &\quad (succ \circ f_{2,1} \circ (pred \circ (x_1^2), x_2^2)) \bullet (\alpha \bullet \perp)) \\
 &= if \circ (x_1^2, 0, x_2^2, succ \circ ((f_{2,1} \circ pred \circ (x_1^2), x_2^2)) \bullet (\alpha \bullet \perp)) \\
 &= if \circ (x_1^2, 0, x_2^2, succ \circ (\alpha \bullet \perp) \circ (pred \circ (x_1^2), x_2^2)) \\
 &= if \circ (x_1^2, 0, x_2^2, succ \circ if \circ ((x_1^2, 0, x_2^2, succ \circ \perp) \\
 &\quad \circ (pred \circ (x_1^2), x_2^2))) \\
 &= if \circ (x_1^2, 0, x_2^2, succ \circ if \circ (pred \circ x_1^2, 0, x_2^2, succ \circ \perp)),
 \end{aligned}$$

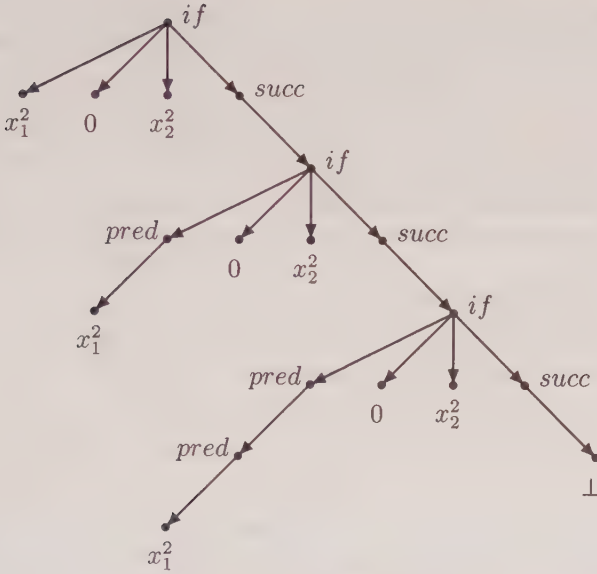
corresponding to the tree



The next step gives us

$$\begin{aligned}
 \alpha^{(2)} &= \alpha^3 \bullet \perp \\
 &= \alpha \bullet (\alpha^2 \bullet \perp) \\
 &= if \circ (x_1^2, 0, x_2^2, succ \circ if \circ (pred \circ x_1^2, 0, x_2^2, \\
 &\quad succ \circ if \circ (pred \circ pred \circ x_1^2, 0, x_2^2, succ \circ \perp)))
 \end{aligned}$$

corresponding to the tree



and it is easy to see intuitively that  $\sqcup \alpha^{(k)}$  is the desired unwinding of the recursion equation represented by  $\alpha$ .

**Example 4.4.4.** As our final example we look further at the context-free grammar discussed in Example 1.2.3.

Let  $\Gamma$  be the alphabet

$$\Gamma =$$

$\{0, \text{succ}, +, *, \text{if}, \text{then}, \text{else}, \text{true}, \text{false}, \text{is\_zero}, \neg, \wedge, \text{while}, \text{do}, :=, ;, (, )\}$ ,

and let  $I$  be a set disjoint from  $\Gamma$ , and let  $A = \mathcal{P}((I \cup \Gamma)^*)$ , the set of all subsets of  $(I \cup \Gamma)^*$ . Recall, from Example 2.4.8, the subtheory  $\mathbf{CF}_G$  of  $\mathbf{Pow}_A$  that we constructed from the context-free grammar:

$$\begin{aligned} \langle \text{st} \rangle &::= (\langle \text{a-id} \rangle := \langle \text{ae} \rangle) \mid (\langle \text{b-id} \rangle := \langle \text{be} \rangle) \mid (\text{if } \langle \text{be} \rangle \text{ then } \langle \text{st} \rangle \text{ else } \langle \text{st} \rangle) \\ &\quad \mid \langle \text{st} \rangle ; \langle \text{st} \rangle \mid (\text{while } \langle \text{be} \rangle \text{ do } \langle \text{st} \rangle) \\ \langle \text{ae} \rangle &::= 0 \mid \langle \text{a-id} \rangle \mid \text{succ}(\langle \text{ae} \rangle) \mid (\langle \text{ae} \rangle + \langle \text{ae} \rangle) \mid (\langle \text{ae} \rangle * \langle \text{ae} \rangle) \\ &\quad \mid (\text{if } \langle \text{be} \rangle \text{ then } \langle \text{ae} \rangle \text{ else } \langle \text{ae} \rangle) \\ \langle \text{be} \rangle &::= \text{true} \mid \text{false} \mid \langle \text{b-id} \rangle \mid \text{is\_zero}(\langle \text{ae} \rangle) \mid \neg(\langle \text{be} \rangle) \mid (\langle \text{be} \rangle \wedge \langle \text{be} \rangle) \end{aligned}$$

from Example 1.2.3. This set of productions can be represented as a morphism  $\alpha : 3 + 2 \rightarrow 3$  in  $\mathbf{CF}_G$ . Using the signature given in Example 2.4.8, and writing the operator ' $\mid$ ' as an infix operator, then  $\alpha$  is the triple of mappings:

$$\begin{aligned} \alpha_1 &= id(x_4^5, x_2^5) \mid ife(x_3^5, x_1^5, x_1^5) \mid com(x_1^5, x_1^5) \mid whl(x_3^5, x_1^5) \\ \alpha_2 &= zero \mid x_4^5 \mid suc(x_2^5) \mid add(x_2^5, x_2^5) \mid mult(x_2^5, x_2^5) \mid ife(x_3^5, x_2^5, x_2^5) \\ \alpha_3 &= true \mid false \mid x_5^5 \mid isz(x_3^5) \mid not(x_3^5) \mid and(x_3^5, x_3^5). \end{aligned}$$

We claim that the abstract flowchart  $\langle \beta, \alpha \rangle$  with  $\beta = x_1^5$  gives the desired semantics for the non-terminal symbol  $\langle \text{st} \rangle$  of ‘statements’. That is,  $\beta \bullet \alpha^\nabla : 2 \rightarrow 1$  is a function

$$(\beta \bullet \alpha^\nabla) : \mathcal{P}((I \cup \Gamma)^*) \times \mathcal{P}((I \cup \Gamma)^*) \rightarrow \mathcal{P}((I \cup \Gamma)^*)$$

which, given a pair of arguments  $A, B \subseteq I^*$ , interpreted respectively as arithmetic-identifiers and boolean-identifiers, will have, as value, the set of all statements produced by the grammar for this choice of identifiers.

## 4.5 Interpretations of flowcharts

It is all very well to unwind the uninterpreted flowchart and get the correct infinite tree, but our real interest in programming is in interpreted flowcharts. How do they fit into the present context?

Our claim is that for any ‘flowchart’  $\langle \beta, \alpha \rangle$  in an iteration theory  $\mathbf{T}$  any reasonable interpretation of  $\langle \beta, \alpha \rangle$  can be represented by a iteration theory morphism  $F : \mathbf{T} \rightarrow \mathbf{T}_{int}$  where  $\mathbf{T}_{int}$  is some appropriate iteration theory. Intuitively, the theory  $\mathbf{T}$  provides a syntactic representation of the flowchart, the theory  $\mathbf{T}_{int}$  provides the desired semantic environment and the functor  $F$  connects the two, that is, it gives the meaning for the syntactic objects of  $\mathbf{T}$ .

There is an intuitive connection between freeness and syntax. We expect the syntax of a program to be ‘writable’ and, loosely speaking, this means that we are going to write it down in a manner such that at least the abstract syntax can be thought of as terms, or trees, over some signature  $\Sigma$ . Indeed, as shown, for example, in [Thatcher *et al.*, 1981], any context-free grammar gives rise to a corresponding free algebra which can be used as the basis for producing the corresponding free theory. On the other hand, we often take advantage of properties such as associativity or commutativity, where they occur, to simplify the syntax. Be that as it may, the theory,  $\mathbf{T}$ , in which the flowcharts are written is generally freely generated by an appropriate signature  $\Sigma$  in the sense that the desired semantic functor  $F : \mathbf{T} \rightarrow \mathbf{T}_{int}$  is completely determined by a signature morphism  $\Sigma \rightarrow \mathbf{T}_{int}$ .

While we want to support this claim by presenting some examples we start by giving a fundamental, if simple, result which provides strong theoretical justification (or, at least, motivation) for the claim. This result is generally referred to as the *Mezei-Wright theorem* in honor of the important special case presented early-on by Mezei and Wright [Mezei and Wright, 1967]. This theorem, which is frequently presented in the form of the slogan

**solve and interpret equals interpret and solve**

goes as follows:

**Theorem 4.5.1.** *Let  $F : \mathbf{T} \rightarrow \mathbf{T}_{int}$  be a morphism of  $S$ -sorted iteration theories and let  $\langle \beta, \alpha \rangle$  be a flowchart in  $\mathbf{T}$ , with  $\alpha : u + v \rightarrow u$ . Then, to*



expand the above slogan, the result of solving  $\langle \beta, \alpha \rangle$  in  $\mathbf{T}$  and then using  $F$  to interpret that solution in  $\mathbf{T}_{int}$  is the same as the result of first using  $F$  to interpret  $\alpha$  and  $\beta$  in  $\mathbf{T}$  and then solving that interpreted flowchart,  $\langle F(\beta), F(\alpha) \rangle$ , in  $\mathbf{T}_{int}$ . More formally:

$$F(\beta \bullet \alpha^\nabla) = F(\beta) \bullet F(\alpha)^\nabla.$$

**Proof.** This is really an immediate consequence of the definitions:

$$\begin{aligned} F(\beta \bullet \alpha^\nabla) &= F(\beta \bullet (\alpha^\dagger, 1_v)) && \text{def } \nabla \\ &= F(\beta) \bullet F((\alpha^\dagger, 1_v)) && F \text{ preserves composition} \\ &= F(\beta) \bullet (F(\alpha^\dagger), 1_v) && F \text{ preserves tupling and identities} \\ &= F(\beta) \bullet (F(\alpha)^\dagger, 1_v) && F \text{ preserves } \dagger \\ &= F(\beta) \bullet F(\alpha)^\nabla && \text{def } \nabla \end{aligned}$$

which completes the proof. ■

Let us now look at some examples:

**Example 4.5.2.** We begin by looking at interpretations of  $\Sigma$ -flowcharts of the type discussed in Examples 1.2.1 and 4.4.2, where the signature  $\Sigma$  is 1-sorted with  $\Sigma_1 = \{F_1, F_2, \dots\}$ ,  $\Sigma_2 = \{P_1, P_2, \dots\}$ , and  $\Sigma_n = \emptyset$  for all  $n \neq 1, 2$ . Working backwards, what we are generally interested in, in a typical flowchart, are interpretations in which each  $F_i$  is interpreted as a state transformation, where a state is, in the simplest case, an assignment of values, say elements of the set  $\mathbf{N}$  of natural numbers, to elements of a set  $Id = \{A, B, \dots\}$  of identifiers. In this case we might expect the state transformations to consist of assignment statements, such as  $A := 2$  or  $B := A * B$  built up from identifiers, natural numbers, and arithmetic operations. Each  $P_i$ , on the other hand, would be considered a conditional operation, such as  $B = C?$  or  $A > 0?$ , which determines, ‘which way to branch’, as a function of the state, but does not necessarily change the state. This is just a special case then of a theory  $\mathbf{Sum}_A$  as described in Example 2.4.2. That is, given that  $Id$  is the set of identifiers, then  $A = \mathbf{N}^{Id}$  is the set of states, and the interpretation is given by mapping each  $F_i$  to a function  $A \rightarrow A$  corresponding to an assignment statement, and each  $P_j$  to an appropriate conditional  $A \rightarrow A \times [2]$  (see Example 2.4.2). This then gives us an interpretation  $f^\sharp : \mathbf{CT}_\Sigma \rightarrow \mathbf{Sum}_A$  by the freeness of  $\mathbf{CT}_\Sigma$ . See the end of section 6 for a bit more on this subject.

**Example 4.5.3.** The theory  $\mathbf{Rec}_\Sigma$  of Examples 1.2.2 and 4.4.3 is also freely generated by  $\Sigma$ . Taking the specific choice for  $\Sigma$  given at the end of Example 1.2.2 there is an obvious interpretation in  $\mathbf{Pow}_\mathbf{N}$ , where  $\mathbf{N}$  denotes the natural numbers, and  $\mathbf{Pow}_\mathbf{N}$  is as defined in Example 2.4.7, that is,  $\mathbf{Pow}_\mathbf{N}(n, p)$  consists of all partial functions  $f : \mathbf{N}^n \rightarrow \mathbf{N}^p$ .

**Example 4.5.4.** In Example 2.4.8 we gave a theory  $\mathbf{CF}_G$  corresponding to a particular context-free grammar. In that case the syntax and semantics

are in the same theory. One might object, however, that  $\mathbf{CF}_G$  is not very syntactic since its morphisms are functions on the set  $(I \cup \Gamma)^*$ . But clearly an appropriate ‘free-er’ theory could be developed from the signature corresponding to the productions. If this is done properly, the resulting theory could be interpreted not only in  $\mathbf{CF}_G$ , but also in a way so that its semantics would be the abstract syntax, i.e., it would define sets of derivation trees rather than sets of strings from the language.

## 5 More about iteration operators

### 5.1 Relationships between iteration, rational, and iterative theories

What is the relationship between iteration, rational, and iterative algebraic theories? The basic answer is that all rational theories are iteration theories, that the ideal morphisms in an iterative theory satisfy the identities for iteration theories, and that every iterative theory can be extended to a rational theory. The proofs of these results are sometimes more lengthy than interesting but we present them to illustrate the forms of proofs found in the three approaches.

We start by presenting an alternative to Axiom 4.2.2.4 from the definition of rational theory.

**Lemma 5.1.1.** *In the definition of rational theory, Axiom 4.2.2.4, i.e.,*

$$(\alpha \bullet (1_u \times \tau))^\dagger = \alpha^\dagger \bullet \tau$$

*for all  $\alpha : u + v \rightarrow u$  and  $\tau : w \rightarrow v$ , can be replaced by the following quantified over all  $\alpha : u + v \rightarrow u$ ,  $\eta : v \rightarrow u$  and  $\tau : w \rightarrow v$ .*

$$\alpha \bullet (\eta, \tau) = \eta \text{ implies } \alpha^\dagger \bullet \tau \sqsubseteq \eta.$$

**Proof.** We first prove (4.2.2.4) implies the above. Assume  $\alpha \bullet (\eta, \tau) = \eta$ , then  $\eta$  is a solution for  $\alpha \bullet (1_u \times \tau)$  since  $\alpha \bullet (1_u \times \tau) \bullet (\eta, 1_w) = \alpha \bullet (\eta, \tau) = \eta$  by (2.5.2.5). But then, by (4.2.2.4),  $(\alpha \bullet (1_u \times \tau))^\dagger = \alpha^\dagger \bullet \tau$ , so, by (4.2.2.3),  $\alpha^\dagger \bullet \tau \sqsubseteq \eta$ , as desired.

For the other direction, assume  $\alpha \bullet (\eta, \tau) = \eta$  implies  $\alpha^\dagger \bullet \tau \sqsubseteq \eta$ . Now  $\alpha^\dagger \bullet \tau$  is a solution for  $\alpha \bullet (1_u \times \tau)$  because

$$\alpha \bullet (1_u \times \tau) \bullet (\alpha^\dagger \bullet \tau, 1_w) = \alpha \bullet (\alpha^\dagger \bullet \tau, \tau) = \alpha \bullet (\alpha^\dagger, 1_v) \bullet \tau = \alpha^\dagger \bullet \tau$$

by (2.5.2.5), (2.5.1.4) and (4.2.2.2) respectively. But if  $\eta$  is any solution for  $\alpha \bullet (1_u \times \tau)$ , then

$$\eta = \alpha \bullet (1_u \times \tau) \bullet (\eta, 1_w) = \alpha \bullet (\eta, \tau),$$

so, from the above assumption,  $\alpha^\dagger \bullet \tau \sqsubseteq \eta$ . Then, in particular,  $\alpha^\dagger \bullet \tau \sqsubseteq (\alpha \bullet (1_u \times \tau))^\dagger$ , so  $\alpha^\dagger \bullet \tau$  is the minimum solution. ■

Using the above lemma we can show that the ‘key identity’ for iteration theories (Axiom 4.2.1.2) holds in iterative and rational theories (assuming, in the iterative case, that the morphisms in question are ideal).

**Proposition 5.1.2.** *In any iterative or rational theory  $\mathbf{T}$ , if  $\alpha : u+v+w \rightarrow u$  and  $\beta : u+v+w \rightarrow v$ , and if, in the iterative case,  $\alpha$  and  $\beta$  are ideal morphisms, then*

$$(\alpha, \beta)^\dagger = (\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla, (\beta \bullet \alpha^\nabla)^\dagger).$$

**Proof.** We first show that the right-hand side is a solution for  $(\alpha, \beta)$ .

$$\begin{aligned}
 & (\alpha, \beta) \bullet ((\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla, (\beta \bullet \alpha^\nabla)^\dagger), 1_w) \\
 &= (\alpha, \beta) \bullet (\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla, ((\beta \bullet \alpha^\nabla)^\dagger, 1_w)) \quad (2.5.1.2) \\
 &= (\alpha, \beta) \bullet (\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla, (\beta \bullet \alpha^\nabla)^\nabla) \quad (4.3.1) \\
 &= (\alpha, \beta) \bullet (\alpha^\dagger, 1_{v+w}) \bullet (\beta \bullet \alpha^\nabla)^\nabla \quad (2.5.1.4) \\
 &= (\alpha \bullet (\alpha^\dagger, 1_{v+w}), \beta \bullet (\alpha^\dagger, 1_{v+w}))(\beta \bullet \alpha^\nabla)^\nabla \quad (2.5.1.4) \\
 &= (\alpha^\dagger, (\beta \bullet \alpha^\nabla)) \bullet (\beta \bullet \alpha^\nabla)^\nabla \quad (4.2.2.2, 4.3.1) \\
 &= (\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla, (\beta \bullet \alpha^\nabla) \bullet (\beta \bullet \alpha^\nabla)^\nabla) \quad (2.5.1.4) \\
 &= (\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla, (\beta \bullet \alpha^\nabla)^\dagger). \quad (4.3.2)
 \end{aligned}$$

This is enough to prove the result for an iterative theory providing  $\alpha$  and  $\beta$  are ideal. However for rational theories we must establish that this is the minimal solution.

To see that it is the minimal solution, let  $(\gamma_0, \gamma_1)$  be any solution, i.e., assume  $(\alpha, \beta) \bullet ((\gamma_0, \gamma_1), 1_w) = (\gamma_0, \gamma_1)$ . Then, by the monotonicity of tupling (3.1.1.3), it is sufficient to prove the two inequalities,  $\alpha^\dagger \bullet (\beta \bullet \alpha^\nabla)^\nabla \sqsubseteq \gamma_0$  and  $(\beta \bullet \alpha^\nabla)^\dagger \sqsubseteq \gamma_1$ , and apply Axiom 4.2.2.3. From the assumption that  $(\gamma_0, \gamma_1)$  is a solution for  $(\alpha, \beta)$ , and from (2.5.1.1) and (2.5.1.2), we get

$$\begin{aligned}
 (5.1.2.2) \quad & \alpha \bullet (\gamma_0, (\gamma_1, 1_w)) \\
 &= \alpha \bullet ((\gamma_0, \gamma_1), 1_w) \\
 &= x_{(1)}^{u+v} \bullet (\alpha, \beta) \bullet ((\gamma_0, \gamma_1), 1_w) \\
 &= x_{(1)}^{u+v} \bullet (\gamma_0, \gamma_1) \\
 &= \gamma_0
 \end{aligned}$$

and, similarly,

$$(5.1.2.3) \quad \beta \bullet (\gamma_0, (\gamma_1, 1_w)) = \gamma_1.$$

Now applying Proposition 5.1.1 to (5.1.2.2) gives

$$(5.1.2.4) \quad \alpha^\dagger \bullet (\gamma_1, 1_w) \sqsubseteq \gamma_0.$$

Replacing  $\gamma_0$  by  $\alpha^\dagger \bullet (\gamma_1, 1_w)$  on the left side of (5.1.2.3) yields

$$(5.1.2.5) \quad \beta \bullet (\alpha^\dagger \bullet (\gamma_1, 1_w), (\gamma_1, 1_w)) \sqsubseteq \gamma_1.$$

by monotonicity (3.1.1.2).

The left side of (5.1.2.5) simplifies to give  $(\beta \bullet \alpha^\nabla) \bullet (\gamma_1, 1_w) \sqsubseteq \gamma_1$ , from which  $(\beta \bullet \alpha^\nabla)^\dagger \sqsubseteq \gamma_1$  follows by (4.2.2.3). Plugging  $(\beta \bullet \alpha^\nabla)$  in for  $\gamma_1$  in (5.1.2.4) gives the other required inequality. ■

**Proposition 5.1.3.** *Axiom 4.2.1.3 for iteration theories holds in rational and iterative theories. That is, if  $\alpha : v \rightarrow u$  (and, in the iterative case,  $\alpha$  is an ideal morphism) then  $(0_u \times \alpha)^\dagger = \alpha$ .*

**Proof.** We start with the iterative case. We leave to the reader the easy exercise of showing that  $\alpha$  ideal implies  $(0_u \times \alpha)$  ideal. It remains then only to show that  $\alpha$  is a solution for  $(0_u \times \alpha)$

$$\begin{aligned} (0_u \times \alpha)(\alpha, 1_v) &= (0_u \bullet \alpha, \alpha) & (2.5.2.5) \\ &= (0_v, \alpha) & (2.5.1.5) \\ &= \alpha. & (2.5.1.3) \end{aligned}$$

For the rational case it is easy to prove, by induction on  $k$ , that  $(0_u \times \alpha)^{(k)} = \alpha$ , for all  $k$ . The desired result follows immediately from (4.2.2.1). ■

**Proposition 5.1.4.** *Axiom 4.2.1.4 for iteration theories holds in rational and iterative theories. That is, if  $\alpha : u + v \rightarrow u$  (and, in the iterative case,  $\alpha$  is an ideal morphism) then  $(\alpha \times 0_w)^\dagger = (\alpha^\dagger \times 0_w)$ .*

**Proof.** We again leave it to the reader to show that  $\alpha$  ideal implies  $(\alpha \times 0_w)$  ideal. That  $(\alpha^\dagger \times 0_w)$  is a solution for  $(\alpha \times 0_w)$  can be shown as follows:

$$\begin{aligned} (\alpha \times 0_w) \bullet ((\alpha^\dagger \times 0_w), 1_{v+w}) &= (\alpha \times 1_\lambda) \bullet (1_{u+v} \times 0_w) \bullet ((\alpha^\dagger \times 0_w), 1_{v+w}) & (2.5.2.6, 2.5.2.7) \\ &= (\alpha \times 1_\lambda) \bullet (1_u \times (1_v \times 0_w)) \bullet ((\alpha^\dagger \times 0_w), 1_{v+w}) & (2.5.2.3, 2.5.2.4) \\ &= (\alpha \times 1_\lambda) \bullet ((\alpha^\dagger \times 0_w), (1_v \times 0_w)) & (2.5.2.5) \\ &= (\alpha \times 1_\lambda) \bullet ((\alpha^\dagger \times 1_\lambda) \bullet (1_v \times 0_w), (1_v \times 0_w)) & (2.5.2.7, 2.5.2.6) \\ &= (\alpha \times 1_\lambda) \bullet ((\alpha^\dagger \times 1_\lambda), 1_v) \bullet (1_v \times 0_w) & (2.5.1.4) \\ &= \alpha \bullet (\alpha^\dagger, 1_v) \bullet (1_v \times 0_w) & (2.5.2.7) \\ &= \alpha^\dagger \bullet (1_v \times 0_w) & \text{def. } \dagger \\ &= (\alpha^\dagger \times 1_\lambda) \bullet (1_v \times 0_w) & (2.5.2.7) \\ &= (\alpha^\dagger \times 0_w). & (2.5.2.6) \end{aligned}$$

That completes the iterative case. To prove the rational case we show, by induction on  $k$ , that  $(\alpha \times 0_w)^{(k)} = (\alpha^{(k)} \times 0_w)$  for all  $k \in \omega$ . For  $k = 0$  we have

$$\begin{aligned}
& (\alpha \times 0_w)^{(0)} \\
&= (\alpha \times 0_w) \bullet (\perp_{v+w, u}, 1_{v+w}) \quad (4.2.2) \\
&= \alpha \bullet (\perp_{v+w, u}, x_{(1)}^{v+w}) \quad (2.5.1.6, 2.5.2.6, 2.5.2.5, 2.5.1.5, 2.5.1.3) \\
&= \alpha \bullet (\perp_{v, u}, x_{(1)}^{v+w}, x_{(1)}^{v+w}) \quad \text{by strictness (3.1.1.1)} \\
&= \alpha \bullet (\perp_{v, u}, 1_v) \bullet x_{(1)}^{v+w} \quad (2.5.1.4) \\
&= \alpha^{(0)} \bullet x_{(1)}^{v+w} \quad (4.2.2) \\
&= \alpha^{(0)} \times 0_w. \quad (2.5.2.1, 2.5.2.7, 2.5.2.6)
\end{aligned}$$

For the induction, assume the result for  $k$  and consider  $k+1$ , then

$$\begin{aligned}
& (\alpha \times 0_w)^{(k+1)} \\
&= (\alpha \times 0_w) \bullet ((\alpha \times 0_w)^{(k)}, 1_{v+w}) \quad (4.2.3) \\
&= (\alpha \times 0_w) \bullet ((\alpha^{(k)} \times 0_w), 1_{v+w}) \quad \text{ind. hyp.} \\
&= (\alpha \times 0_w) \bullet (((\alpha^{(k)} \times 0_w), x_{(1)}^{v+w}), x_{(2)}^{v+w}) \quad (2.5.1.6, 2.5.1.2) \\
&= (\alpha \times 0_w) \bullet ((\alpha^{(k)} \times 1_\lambda) \bullet (1_v \times 0_w), x_{(1)}^{v+w}, x_{(2)}^{v+w}) \quad (2.5.2.7, 2.5.2.6) \\
&= (\alpha \times 0_w) \bullet ((\alpha^{(k)} \times 1_\lambda), 1_v) \bullet x_{(1)}^{v+w}, x_{(2)}^{v+w} \quad (2.5.2.1, 2.5.1.4) \\
&= (\alpha \bullet (\alpha^{(k)}, 1_v) \bullet x_{(1)}^{v+w}, 0_w \bullet x_{(2)}^{v+w}) \quad (2.5.2.7, 2.5.2.5) \\
&= (\alpha^{(k+1)} \bullet x_{(1)}^{v+w}, 0_w \bullet x_{(2)}^{v+w}) \quad (4.2.3) \\
&= (\alpha^{(k+1)} \times 0_w) \quad (\text{def. } \times)
\end{aligned}$$

■

Finally, we show that Axiom 4.2.1.5 for iteration theories holds for iterative and rational theories. This, together with (5.1.2, 5.1.3, 5.1.4), shows that rational and iterative theories are iteration theories.

**Proposition 5.1.5.** *In any iterative or rational theory, if  $u, w \in S^*$ , and  $v \in S^m$ ,  $\beta : v + w \rightarrow u$ , and  $f : u \rightarrow v$ , and  $g_1, \dots, g_m : v \rightarrow v$ , are base morphisms such that  $f$  is surjective and  $g_i \bullet f = f$  for each  $i \in [m]$ , then*

$$(x_1^v \bullet f \bullet \beta \bullet (g_1 \times 1_w), \dots, x_m^v \bullet f \bullet \beta \bullet (g_m \times 1_w))^\dagger = f \bullet (\beta \bullet (f \times 1_w))^\dagger.$$

**Proof.** We will only give the full proof for the case of iterative theories. That is, we will establish that  $f \bullet (\beta \bullet (f \times 1_w))^\dagger$  is a fixpoint for  $(x_1^v \bullet f \bullet \beta \bullet (g_1 \times 1_w), \dots, x_m^v \bullet f \bullet \beta \bullet (g_m \times 1_w))$ , and so the desired result holds for all ideal  $\beta$  in any iterative theory **T**.

Let  $\gamma = (x_1^v \bullet f \bullet \beta \bullet (g_1 \times 1_w), \dots, x_m^v \bullet f \bullet \beta \bullet (g_m \times 1_w))$ . What we must show is that  $\gamma \bullet (f \bullet (\beta \bullet (f \times 1_w))^\dagger, 1_w) = f \bullet (\beta \bullet (f \times 1_w))^\dagger$ . By the definition of algebraic theory, (2.2.2.2) and (2.5.1.4), it suffices to show that  $x_i^v \bullet \gamma \bullet (f \bullet (\beta \bullet (f \times 1_w))^\dagger, 1_w) = x_i^v \bullet (f \bullet (\beta \bullet (f \times 1_w))^\dagger)$  for each  $i = 1, \dots, m$ . Now



$$\begin{aligned}
& x_i^v \bullet \gamma \bullet (f \bullet (\beta \bullet (f \times 1_w))^\dagger, 1_w) \\
&= (x_i^v \bullet f \bullet \beta \bullet (g_i \times 1_w)) \bullet (f \bullet (\beta \bullet (f \times 1_w))^\dagger, 1_w) \quad \text{def. } \gamma \\
&= x_i^v \bullet f \bullet \beta \bullet (g_i \bullet f \bullet (\beta \bullet (f \times 1_w))^\dagger, 1_w) \quad (2.5.2.5) \\
&= x_i^v \bullet f \bullet \beta \bullet (f \bullet (\beta \bullet (f \times 1_w))^\dagger, 1_w) \quad g_i \bullet f = f \\
&= x_i^v \bullet f \bullet \beta \bullet ((f \times 1_w) \bullet ((\beta \bullet (f \times 1_w))^\dagger, 1_w)) \quad (2.5.2.5) \\
&= x_i^v \bullet f \bullet (\beta \bullet (f \times 1_w))^\dagger, \quad \text{def. } \dagger
\end{aligned}$$

which is what we wanted to show. ■

The next result shows that we can take an iterative theory and replace the partial iteration operator by a total iteration operator. We give the result only for the 1-sorted case since this simplifies the induction, but the theorem can be generalized to arbitrary  $S$ -sorted iterative theories.

**Proposition 5.1.6.** ([Bloom et al., 1980b]) *Let  $\mathbf{T}$  be a 1-sorted iterative theory. Let  $\perp : 0 \rightarrow 1$  be a morphism in  $\mathbf{T}$ . Then for all  $n, p \in \omega$ , there is exactly one operator*

$$\begin{aligned}
\uparrow_{n,p} : \mathbf{T}(n+p, n) &\rightarrow \mathbf{T}(p, n) \\
\gamma &\mapsto \gamma^\dagger,
\end{aligned}$$

such that

$$(5.1.6.1) \quad (1_1 \times 0_p)^\dagger = (1_1)^\dagger \times 0_p,$$

$$(5.1.6.2) \quad (1_1)^\dagger = \perp,$$

$$(5.1.6.3) \quad \alpha^\dagger = \alpha \bullet (\alpha^\dagger, 1_p) \text{ holds for all } \alpha : 1+p \rightarrow 1, p \geq 0,$$

and,

$$\begin{aligned}
(5.1.6.4) \quad & (\alpha, \beta)^\dagger = (\alpha^\dagger \bullet ((\beta \bullet (\alpha^\dagger, 1_{1+p}))^\dagger, 1_p), (\beta \bullet (\alpha^\dagger, 1_{1+p}))^\dagger) \\
& \text{holds for all } \alpha : n+1+p \rightarrow n \text{ and } \beta : n+1+p \rightarrow 1.
\end{aligned}$$

**Proof.** Let us first restrict our attention to morphisms  $\gamma : 1+p \rightarrow 1$ . If  $\gamma$  is ideal then let  $\gamma^\dagger$  be the unique morphism satisfying (5.1.6.3) (thus on such ideal morphisms the extended and the unextended iteration operations agree). If  $\gamma : 1+p \rightarrow 1$  is not ideal then it must be a base morphism and so must be of the form  $x_i^{1+p} : 1+p \rightarrow 1$  for some  $i \in \{1, \dots, 1+p\}$ . If  $i = 1$ , so  $\gamma = x_1^{1+p}$ , then define  $\gamma^\dagger$  using (5.1.6.1), (5.1.6.2) and (2.5.2.1). If  $i > 1$ , then  $\gamma^\dagger = x_{i-1}^{1+p}$  is easily seen to be the unique solution to (5.1.6.3) since for any  $\alpha : p \rightarrow 1$ ,  $x_i^{1+p} \bullet (\alpha, 1_p) = x_{i-1}^{1+p}$  if  $1 < i \leq 1+p$ . Thus we have a unique solution for all  $\gamma : 1+p \rightarrow 1$ .

Now if  $\gamma : 0+p \rightarrow 0$ , then  $\gamma = 0_p$ , and trivially,  $\gamma^\dagger = 0_p$ .

To define  $\gamma^\dagger$  for  $\gamma : n+p \rightarrow n$  where  $n > 1$  we proceed by induction on  $n$ . Assume that for all  $p$  and for all  $\delta : k+p \rightarrow k$ ,  $k \leq n$ ,  $\delta^\dagger$  is defined. If (as in (5.1.6.4))  $\gamma = (\alpha, \beta) : n+1+p \rightarrow n+1$  with  $\alpha : n+1+p \rightarrow n$  and  $\beta : n+1+p \rightarrow 1$  then define  $\gamma^\dagger$  by (5.1.6.4).

It is then obvious that (5.1.6.4) holds for all  $n, p \geq 0$ , noting that when  $n = 0$ , then  $\alpha : 0+1+p \rightarrow 0$ , forcing  $\alpha = 0_{1+p}$ , and  $\beta : 0+1+p \rightarrow 1$ , so (5.1.6.4) becomes

$$\begin{aligned}
(0_{p+1}, \beta)^\dagger &= (0_{1+p}^\dagger \bullet (((\beta \bullet (0_{1+p}^\dagger))^\dagger, 1_p), (\beta \bullet (0_{1+p}^\dagger, 1_{1+p})))^\dagger) \\
&= (0_{1+p}, (\beta \bullet (0_{1+p}, 1_{1+p}))^\dagger) \\
&= (0_{1+p}, \beta^\dagger)
\end{aligned}$$

which is trivially true. ■

Starting from the above, we can, with effort, show the following.

**Corollary 5.1.7.** *Every iterative theory can be extended to an iteration theory.*

**Proposition 5.1.8.** *There exists an iteration theory which is not an iterative theory, indeed it is not even ideal. In particular,  $\mathbf{Sum}_A$  is such a theory.*

**Proposition 5.1.9.** *([Bloom et al., 1980b]) There exists an ideal iteration theory which is not an iterative theory.*

**Proof.** Let  $S$  be the set consisting of the natural numbers  $\omega = \{0, 1, 2, \dots\}$  and two additional elements  $a$  and  $b$ , where  $a \neq b$ . Let  $g : S \rightarrow S$  such that  $g(n) = n + 1$  for  $n \in \omega$ ,  $g(a) = a$  and  $g(b) = b$ . Let  $P$  be the smallest subtheory of  $\mathbf{Pow}_S$  containing  $g : 1 \rightarrow 1$ ,  $a : 0 \rightarrow 1$  and  $b : 0 \rightarrow 1$ .

The theory  $P$  is easily seen to be an ideal theory which is not iterative, since the iteration equation for  $g$  has two solutions.

Now define  $g^\dagger = a = (1)^\dagger$  in  $P$  and extend  $\dagger$  to all other morphism as in (5.1.6) getting a new theory  $\bar{P}$ . We claim that  $\bar{P}$  is the desired example. ■

We end this subsection with a brief look at how one can show that there exist iteration theories which can not be made into rational theories. That is, there is no way to order the morphisms so that  $\alpha^\dagger$  will be a least fixpoint.

**Definition 5.1.10.** Let  $\mathbf{T}$  be an algebraic theory, a *compatible partial ordering* on  $\mathbf{T}$  is an  $(S^* \times S^*)$ -indexed family of orderings  $\langle \sqsubseteq_{u,v} \mid u, v \in S^* \rangle$  such that composition and tupling are monotonic with respect to  $\sqsubseteq$ , i.e. such that conditions (3.1.1.2) and (3.1.1.3) of the definition of ordered theory are satisfied.

Since every rational theory is an ordered theory, it is immediate that the ordering on a rational theory is a compatible partial ordering on the theory.

**Definition 5.1.11.** A  $\Sigma$ -tree  $t$  will be said to be *homogeneous* if for each vertex  $v$  of  $t$ , the tree of descendants of  $v$  in  $t$  is isomorphic to  $t$ .

**Proposition 5.1.12.** *(Proposition 5.50, Bloom and Tindell [Bloom and Tindell, 1980]). Let  $\perp : 0 \rightarrow 1$  be a tree in the iterative theory  $\mathbf{T}$  of all  $\Sigma$ -trees. then there is a compatible partial ordering  $\sqsubseteq$  on  $\mathbf{T}$  such that  $(1_1)^\dagger = \perp$  and  $\alpha^\dagger$  is the  $\sqsubseteq$ -least solution of the iteration equation for each  $\alpha$  iff  $\perp$  is homogeneous.*

**Proposition 5.1.13.** *There exists a one-sorted iteration theory  $\mathbf{T}$  for which there is no compatible ordering  $\sqsubseteq$  on  $\mathbf{T}$  such that for every  $\alpha \in \mathbf{T}(u + v, u)$   $\alpha^\dagger$  is always the least fixpoint with respect to  $\sqsubseteq$ .*

**Proof.** Let  $\Sigma$  be a one-sorted signature where  $\Sigma_0$  and  $\Sigma_1$  are non-empty. Recall from Definition 3.2.4 that there is an algebraic theory  $\mathbf{TT}_\Sigma$  in which the morphisms are tuples of total  $\Sigma$ -trees including infinite trees. It can be shown that  $\mathbf{TT}_\Sigma$  is an iterative theory and thus, by Proposition 5.1.6 and Corollary 5.1.7, we can show that for any choice of a morphism  $\perp : 0 \rightarrow 1$  in  $\mathbf{TT}_\Sigma$  there is a unique operator  $\dagger$  such that  $1_1^\dagger = \perp$  and  $\dagger$  extends  $\mathbf{TT}_\Sigma$  to an iteration theory  $\mathbf{TT}_\perp$ . Then, in particular, since  $\Sigma_0, \Sigma_1 \neq \emptyset$ , we can take  $\gamma_0 \in \Sigma_0, \gamma_1 \in \Sigma_1$  and let  $\perp = \gamma_1^\dagger \bullet \gamma_0$  and construct a corresponding iteration theory with  $\mathbf{T}_\perp$  in which  $1_1^\dagger = \perp = \gamma_1 \bullet \gamma_0$ . But  $\perp = \gamma_1 \bullet \gamma_0$  corresponds to the tree



which is clearly not homogeneous. Thus, by Proposition 5.1.12, there is no compatible ordering on  $\mathbf{T}_\perp$  such that  $1_1^\dagger = \perp$ . But in any ordered theory, and thus in any rational theory,  $\perp$  is the least fixpoint for  $1_1$ :

$$1_1 \bullet (\perp, 1_0) = 1_1 \bullet (\perp, 0_0) = \perp.$$

Thus  $\mathbf{TT}_\perp$  is an iteration theory but not a rational theory. ■

## 5.2 Some identities for iteration operators

This section of identities can certainly be skipped the first time through the material. The proposition provides some useful ‘work-horse’ identities which are exercised in the proof of the subsequent lemma. That lemma provides the needed lemmata for the ‘iteration closure’ theorem of the next section.

**Proposition 5.2.1.** *Let  $\mathbf{T}$  be an iteration theory with the following morphisms.*

$$\begin{aligned} \eta : u + v + w \rightarrow u, & \quad \nu : u + v + w \rightarrow v, \\ \alpha : u + v \rightarrow u, & \quad \beta : v + w \rightarrow v, \\ \tau : w \rightarrow v, & \quad \gamma : u + w \rightarrow u. \end{aligned}$$

Then

$$(5.2.1.1) \quad (\eta, \nu)^\nabla = \eta^\nabla \bullet (\nu \bullet \eta^\nabla)^\nabla$$

$$(5.2.1.2) \quad \alpha^\nabla \bullet \tau = (\alpha \times \tau)^\dagger = x_{(1,2)}^{u+v+w} \bullet (\alpha \times \tau)^\nabla$$

$$(5.2.1.3) \quad \eta^\nabla \bullet \beta^\nabla = ((\eta \times \beta) \bullet (1_{u+v+w}, x_{(2,3)}^{u+v+w}))^\nabla$$

$$(5.2.1.4) \quad (\beta^\nabla, \gamma^\nabla) = (x_{(1,3)}^{v+u+w}, x_{(2,3)}^{v+u+w}) \bullet ((\beta \times \gamma) \bullet (x_{(1,3)}^{v+u+w}, x_{(2,3)}^{v+u+w}))^\nabla$$

**Proof.** (5.2.1.1) Pair both sides of (4.2.1.2) with  $1_w$  and apply (4.3.1). (2.5.1.2), (2.5.1.4), and (4.3.1) again.

(5.2.1.2) From the definition of  $\times$ , (2.5), we have  $(\alpha \times \tau)^\dagger = (\alpha \bullet x_{(1,2)}^{u+v+w}, \tau \bullet x_{(3)}^{u+v+w})^\dagger$ , so we want to evaluate the left side of (4.2.1.2) with this pair. The second component simplifies to  $\tau$  (an exercise for the reader). The first component is

$$\begin{aligned} (\alpha \bullet x_{(1,2)}^{u+v+w})^\dagger \bullet (\tau, 1_w) &= (\alpha \bullet (1_u \times x_{(1)}^{v+w}))^\dagger \bullet (\tau, 1_w) \quad (\text{exercise}) \\ &= (\alpha^\dagger \bullet x_{(1)}^{v+w}) \bullet (\tau, 1_w) \quad (4.2.1.6) \\ &= \alpha^\dagger \bullet \tau, \quad (2.5.1.1) \end{aligned}$$

and the right-hand side becomes

$$(\alpha^\dagger \bullet \tau, \tau) = (\alpha^\dagger, 1_v) \bullet \tau = \alpha^\nabla \bullet \tau$$

as required.

(5.2.1.3) Since  $\eta^\nabla = (\eta^\dagger, 1_{v+w})$ , putting  $\beta \bullet x_{(2,3)}^{u+v+w}$  in for  $\nu$  in (5.2.1.1) gives this identity.

(5.2.1.4) By (2.5.2.5)

$$((\beta \times \gamma) \bullet (x_{(1,3)}^{v+u+w}, x_{(2,3)}^{v+u+w}))^\nabla = (\beta \bullet x_{(1,3)}^{v+u+w}, \gamma \bullet x_{(2,3)}^{v+u+w})^\nabla,$$

so we want to evaluate the right-hand side using (5.2.1.1). First:

$$\begin{aligned} (\beta \bullet x_{(1,3)}^{v+u+w})^\nabla &= ((\beta \bullet (1_v \times x_{(2)}^{u+w}))^\dagger, 1_{u+w}) \quad (2.5.2.1), \text{ def } \nabla \\ &= (\beta^\dagger \bullet x_{(2)}^{u+w}, 1_{u+w}). \quad (4.2.1.6) \end{aligned}$$

But  $x_{(2,3)}^{v+u+w} \bullet (\beta^\dagger \bullet x_{(2)}^{u+w}, 1_{u+w}) = 1_{u+w}$  so the right-hand side of (5.2.1.1) becomes

$$(\beta^\dagger \bullet x_{(2)}^{u+w}, 1_{u+w}) \bullet \gamma^\nabla = (\beta^\dagger, \gamma^\dagger, 1_w)$$

by (2.5.1.4) and def.  $\nabla$ . Now pre-multiplying  $(\beta^\dagger, \gamma^\dagger, 1_w)$  by the base morphism  $(x_{(1,3)}^{v+u+w}, x_{(2,3)}^{v+u+w})$  gives  $((\beta^\dagger, 1_w), \gamma^\dagger, 1_w)$  which is  $(\beta^\nabla, \gamma^\nabla)$ . ■

As noted above, the following lemma is a collection of identities that are exactly the ‘sub goals’ in the proof of the ‘iteration closure’ theorem of the next section. Except for the first (and most trivial) they are of little interest in their own right and amount to reformulating previous identities tailored to the specific needs of the rational closure theorem.

**Lemma 5.2.2.** *For any iteration theory  $\mathbf{T}$ , the following identities hold.*

(5.2.2.1)  $(0_u)^\nabla = 1_u$

(5.2.2.2) *For  $\beta_1 : r + v \rightarrow u$  and  $\alpha_1 : r + v \rightarrow r$ ,  $\beta_1 \bullet \alpha_1^\nabla : v \rightarrow u$ ; and for  $\beta_2 : t + w \rightarrow v$  and  $\alpha_2 : t + w \rightarrow t$ ,  $\beta_2 \bullet \alpha_2^\nabla : w \rightarrow v$ . The two ‘behaviours’ are composable and:*

$$\begin{aligned} &(\beta_1 \bullet \alpha_1^\nabla) \bullet (\beta_2 \bullet \alpha_2^\nabla) = \\ &\beta_1 \bullet x_{(1,2)}^{r+v+t+w} \bullet ((\alpha_1 \times \beta_2 \times \alpha_2) \bullet (1_{r+v+t+w}, x_{(3,4)}^{r+v+t+w}))^\nabla. \end{aligned}$$

(5.2.2.3) For  $\beta_i : u_i \rightarrow w_i$  and  $\alpha_i : u_i + v \rightarrow u_i$ ,  $i = 1, 2$ , the behaviors,  $\beta_i \bullet \alpha_i^\nabla$  can be paired and:

$$(\beta_1 \bullet \alpha_1^\nabla, \beta_2 \bullet \alpha_2^\nabla) = (\beta_1 \times \beta_2) \bullet f \bullet ((\alpha_1 \times \alpha_2) \bullet f)^\nabla.$$

where  $f = (x_{(1,3)}^{u_1+u_2+v}, x_{(2,3)}^{u_1+u_2+v})$ .

(5.2.2.4) For  $\beta : r + u + v \rightarrow u$  and  $\alpha : r + u + v \rightarrow r$ , the iterate of  $\beta \bullet \alpha^\nabla$  is defined and

$$(\beta \bullet \alpha^\nabla)^\nabla = x_{(2,3)}^{r+u+v} \bullet (\alpha, \beta)^\nabla.$$

**Proof.** (5.2.2.1) By definition  $0_u$  is the unique morphism  $0_u : u \rightarrow \lambda$ . Now

$$\begin{aligned} 0_u^\dagger &= 0_u \bullet (0_u^\dagger, 1_u) \quad \text{by def } \dagger \\ &= 0_u. \quad (2.5.1.5) \end{aligned}$$

But then

$$\begin{aligned} 0_u^\nabla &= (0_u^\dagger, 1_u) \quad \text{def. } \nabla \\ &\Rightarrow (0_u, 1_u) \quad \text{by above} \\ &= 1_u. \quad (2.5.1.3) \end{aligned}$$

(5.2.2.2) We have

$$\begin{aligned} &(\beta_1 \bullet \alpha_1^\nabla) \bullet (\beta_2 \bullet \alpha_2^\nabla) \\ &= \beta_1 \bullet (\alpha_1^\nabla \bullet \beta_2) \bullet \alpha_2^\nabla \quad \bullet \text{ associative} \\ &= \beta_1 \bullet x_{(1,2)}^{r+v+t+w} \bullet (\alpha_1 \times \beta_2)^\nabla \bullet \alpha_2^\nabla \quad (5.2.1.2) \\ &= \beta_1 \bullet x_{(1,2)}^{r+v+t+w} \bullet (((\alpha_1 \times \beta_2) \times \alpha_2) \\ &\quad \bullet (1_{r+v+t+w}, x_{(3,4)}^{r+v+t+w}))^\nabla. \quad (5.2.1.3) \end{aligned}$$

(5.2.2.3) We have

$$\begin{aligned} &(\beta_1 \bullet \alpha_1^\nabla, \beta_2 \bullet \alpha_2^\nabla) \\ &= (\beta_1 \times \beta_2) \bullet (\alpha_1^\nabla, \alpha_2^\nabla) \quad (2.5.2.5) \\ &= (\beta_1 \times \beta_2) \bullet f \bullet ((\alpha_1 \times \alpha_2) \bullet f)^\nabla \quad \text{by (5.2.1.4)} \\ &\text{where } f = (x_{(1,3)}^{u_1+u_2+v}, x_{(2,3)}^{u_1+u_2+v}). \end{aligned}$$

$$\begin{aligned} &(\beta \bullet \alpha^\nabla)^\nabla \\ &= x_{(2,3)}^{r+u+v} \bullet (\alpha^\dagger, 1_{u+v}) \bullet (\beta \bullet \alpha^\nabla)^\nabla \\ &= x_{(2,3)}^{r+u+v} \bullet \alpha^\nabla \bullet (\beta \bullet \alpha^\nabla)^\nabla \\ &= x_{(2,3)}^{r+u+v} \bullet (\alpha, \beta)^\nabla \quad (5.2.1.1) \end{aligned}$$

## 6 Iteration closure, and normal form theorems

We come now to one of the main theorems. Informally, this result says that the ‘behaviours of flow charts’ in an iteration theory themselves form



an iteration theory. Of course, as we have been trying to stress all along, the ‘flowcharts’ in question can be things such as context-free grammars, monadic recursion schemes, recursive definitions, etc. as well as ‘ordinary’ flowcharts. However the flowchart case does have special significance as we will show later on.

**Theorem 6.0.1.** [*Wright et al., 1976*] Let  $\mathbf{F}$  be a subtheory of iteration theory  $\mathbf{T}$  and for each pair  $u, v \in S^*$ , define  $\mathbf{R}(v, u)$  to be the set of all  $\beta \bullet \alpha^\nabla$  such that  $\alpha : w + v \rightarrow w$  and  $\beta : w + v \rightarrow u$  in  $\mathbf{F}$  and  $w \in S^*$ . Then  $\mathbf{R}$  is a subtheory of  $\mathbf{T}$  which is an iteration theory and it is the smallest such containing  $\mathbf{F}$ .

**Proof.**  $\mathbf{R}$  contains  $\mathbf{F}$  for if  $\gamma : u \rightarrow v$  in  $\mathbf{F}$  then  $\gamma \bullet 0_u^\nabla = \gamma \bullet 1_u = \gamma$  by (5.2.2.1). It follows from this that  $\mathbf{R}$  contains all identities  $1_u$  and distinguished morphisms  $x_i^u$ .  $\mathbf{R}$  is closed under composition by (5.2.2.2), and under tupling by (5.2.2.3) (and thus each  $x_{(i)}^{u+v} \in \mathbf{R}$ ). Thus  $\mathbf{R}$  is a subtheory of  $\mathbf{T}$ , and we need only prove it is closed under the iteration operator of  $\mathbf{T}$ . But if  $\gamma \in \mathbf{R}$  ( $\gamma = \alpha \bullet \beta^\nabla$ ) then, by (5.2.2.4),  $\gamma^\nabla (= (\alpha \bullet \beta^\nabla)^\nabla)$  is in  $\mathbf{R}$ , and so  $\gamma^\dagger = x_{(1)}^{u+v} \bullet \gamma^\nabla$  is in  $\mathbf{R}$ . The remaining conditions for  $\mathbf{R}$  to be an iteration theory (see Definition (4.2.1)) are satisfied since  $\mathbf{R}$  is a subtheory of  $\mathbf{T}$  and they hold in  $\mathbf{T}$ . ■

**Definition 6.0.2.** Let  $\mathbf{T}$ ,  $\mathbf{F}$ , and  $\mathbf{R}$  be as in the above theorem, then we call  $\mathbf{R}$  the iteration closure of  $\mathbf{F}$  in  $\mathbf{T}$ .

We can strengthen the above theorem in a number of ways to get various normal form theorems. The following is a fairly straightforward example of such a theorem and can be viewed as a generalization to arbitrary iteration theories of the ‘Chomsky normal form’ theorem for context-free grammars. Other applications are discussed after the proof.

**Theorem 6.0.3.** Let  $\mathbf{T}$  be an iteration theory, let  $F = \langle F_{u,s} \mid F_{u,s} \subseteq \mathbf{T}(u, s), u \in S^*, s \in S \rangle$  be a subfamily of  $\langle \mathbf{T}(u, s) \mid u \in S^*, s \in S \rangle$  such that  $1_s \in F_{s,s}$  for all  $s \in S$ . Let  $F^\times$  be the closure of  $F$  under  $\times$ . Let  $\mathbf{R}(u, v)$  be the set of all  $b \bullet \alpha^\nabla$  such that  $b : w + v \rightarrow u$  is a base morphism,  $\alpha : w + v \rightarrow w$ , and  $\alpha = \alpha' \bullet c$  where  $\alpha' \in F^\times$  and  $c$  is a base morphism. Then  $\mathbf{R}$  is a subtheory of  $\mathbf{T}$  which is an iteration theory and it is the smallest such containing  $F$ .

**Proof.** Since  $1_s \in F_{s,s}$  for each  $s \in S$  it follows that, for each  $u \in S^*$   $1_u \in F^\times$  since if  $u = s_1 \cdots s_n$  then  $1_u = 1_{s_1} \times \cdots \times 1_{s_n} \in F^\times$ .

Claim 1: For every base morphism  $f : u \rightarrow v$ ,  $f \in \mathbf{R}$ .

Clearly, taking  $\alpha' = 1_u \in F^\times$  and  $c = f \bullet_{(2)}^{u+v}$  we have that  $\alpha \bullet c$  is of the required form, but

$$\begin{aligned}
& x_{(1)}^{u+v} \bullet (\alpha' \bullet c)^\nabla \\
&= x_{(1)}^{u+v} \bullet (1_u \bullet f \bullet x_{(2)}^{u+v})^\nabla \\
&= (f \bullet x_{(2)}^{u+v})^\dagger \quad \text{def } \nabla \\
&= f.
\end{aligned}$$

Claim 2. If  $\gamma \in F_{u,s}$  then  $\gamma \in \mathbf{R}$ .

Now  $\gamma : u \rightarrow s$ , but then

$$\gamma = (\gamma \bullet x_{(2)}^{u+s})^\dagger = x_{(1)}^{u+s} \bullet (\gamma \bullet x_{(2)}^{u+s})^\nabla.$$

Claim 3.  $\mathbf{R}$  is closed under composition and tupling.

Given  $b_1 : v + w \rightarrow u$ ,  $\alpha_1 : v + w \rightarrow v$ ,  $b_2 : t + r \rightarrow w$ , and  $\alpha_2 : t + r \rightarrow r$  such that  $b_i \bullet \alpha_i^\nabla \in \mathbf{R}$ ,  $i = 1, 2$ , we have, by (5.2.2.2), that

$$\begin{aligned}
& (b_1 \bullet \alpha_1^\nabla) \bullet (b_2 \bullet \alpha_2^\nabla) \\
&= b_1 \bullet x_{(1,2)}^{v+w+t+r} \bullet [(\alpha_1 \times b_2 \times \alpha_2) \bullet (1_{v+w+r+r}, x_{(3,4)}^{v+w+t+r})]^\nabla \\
&= b_1 \bullet x_{(1,2)}^{v+w+t+r} \bullet [(\alpha_1 \times 1_w \times \alpha_2) \bullet (x_{(1,2)}^{v+w+t+r}, (b_2, 1_{t+r}) \bullet x_{(3,4)}^{v+w+t+r})]^\nabla
\end{aligned}$$

which is easily seen to be of the right form, using the fact that if  $\alpha_i = \alpha' \bullet c_i$ ,  $\alpha' \in F^\times$ ,  $c_i$  base, then

$$(\alpha_1 \times 1_w \times \alpha_2) = (\alpha'_1 \times 1_w \times \alpha'_2) \bullet (c_1 \times 1_w \times c_2).$$

Thus we have closure under composition and closure under tupling is immediate from (5.2.2.3).

Since claim 1 supplies the distinguished morphisms we have that  $\mathbf{R}$  is a subtheory of  $\mathbf{T}$ .

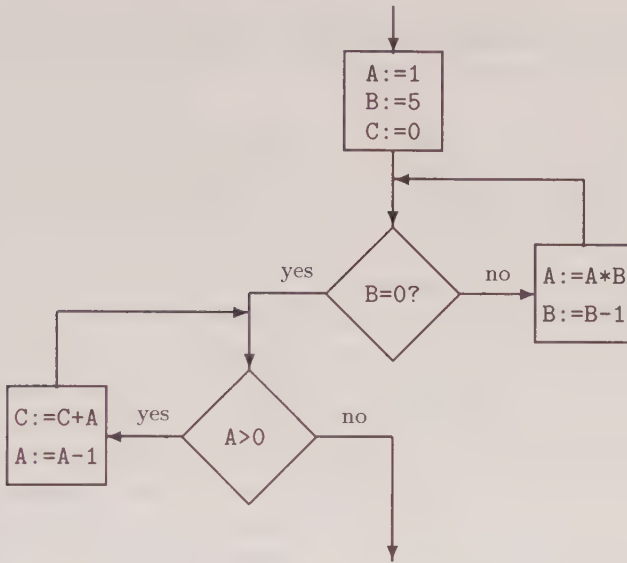
To see that  $\gamma \in \mathbf{R}$  implies  $\gamma^\dagger \in \mathbf{R}$  let  $\gamma = b \bullet (\alpha' \bullet c)^\nabla$  with  $b : u + v + w \rightarrow v$  and  $\alpha' \bullet c : u + v + w \rightarrow u$ , then

$$\begin{aligned}
\gamma^\dagger &= x_{(1)}^{v+w} \bullet (b \bullet (\alpha' \bullet c)^\nabla)^\nabla \quad \text{def } \nabla \\
&= x_{(1)}^{v+w} \bullet (x_{(2,3)}^{u+v+w} \bullet (\alpha' \bullet c, b)^\nabla)^\nabla \quad (5.2.2.4) \\
&= x_{(2)}^{u+v+w} \bullet ((\alpha' \times 1_u) \bullet (c, b))^\nabla \quad (2.5.2.5)
\end{aligned}$$

and the rest of the theorem follows. ■

Elgot's 'normal descriptions' [Elgot, 1975] are an example of the kind of restriction given above. A normal description is essentially a flowchart  $\langle \beta, \alpha \rangle$  where  $\beta$  is restricted to being a base morphism. The above theorem shows that this restriction does not limit the expressive power.

A more familiar example is the way in which we traditionally write programs corresponding to 'ordinary' flowcharts. Consider the following flowchart.



A program corresponding to this flowchart would be something like:

```

[1]  A := 1;
[2]  B := 5;
[3]  C := 0;
[4]  IF B = 0 GOTO 8;
[5]  A := A * B;
[6]  B := B - 1;
[7]  GOTO 4;
[8]  IF A > 0 GOTO 10;
[9]  GOTO 13;
[10] C := C + A;
[11] A := A - 1;
[12] GOTO 8;
[13] EXIT
  
```

which corresponds to the morphism

$$\begin{aligned}
 & ( (A := 1)(x_2^{13}), (B := 5)(x_3^{13}), (C := 0)(x_4^{13}), (B = 0?)(x_8^{13}, x_5^{13}), \\
 & (A := A * B)(x_6^{13}), (B := B - 1)(x_7^{13}), x_4^{13}, (A > 0?)(x_{10}^{13}, x_9^{13}), \\
 & x_{13}^{13}, (C := C + A)(x_{11}^{13}), (A := A - 1)(x_{12}^{13}), x_8^{13} ) : 13 \rightarrow 12
 \end{aligned}$$

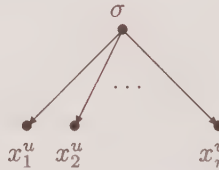
This form of program is not in the normal form given in Theorem 6.0.3, but it is not hard to see that, because of the GOTOs,  $x_i^n$ , we can transform any normal form program on this signature into an equivalent program in this restricted form. It follows then from the normal form theorem

that using this conventional restricted form does not cost us anything in computational power.

## 7 Free theories and Herbrand interpretations

Given an  $S$ -sorted signature  $\Sigma$  we can define the class,  $\Sigma$ -Flow, of all *uninterpreted*  $\Sigma$ -flowcharts as the class of all abstract flowcharts  $\langle \beta, \alpha \rangle$  where for some  $u, v, w \in S^*$ ,  $\alpha : u + v \rightarrow u$  and  $\beta : u + v \rightarrow w$  in  $\mathbf{T}_\Sigma$ . One reason for defining the  $\Sigma$ -flowcharts in  $\mathbf{T}_\Sigma$  is that the morphisms in  $\mathbf{T}_\Sigma$  are tuples of finite  $\Sigma$ -labelled trees (see Example 2.4.5). That is, the abstract flowcharts correspond to finite flowcharts in the informal sense of Section 1. Another reason for defining the  $\Sigma$ -flowcharts in  $\mathbf{T}_\Sigma$  is that  $\mathbf{T}_\Sigma$  is the  $S$ -sorted Lawvere theory freely generated by  $\Sigma$ .

**Proposition 7.0.1.** *Let  $\Sigma$  be an  $S$ -sorted signature, then for any  $S$ -sorted Lawvere theory  $\mathbf{T}$ , and any  $(S^* \times S)$ -indexed family of mappings  $f = \langle f_{u,s} : \Sigma_{u,s} \rightarrow \mathbf{T}(u, s) \mid u \in S^*, s \in S \rangle$  there is a unique theory morphism  $F : \mathbf{T}_\Sigma \rightarrow \mathbf{T}$  that extends  $f$ . Which is to say, for any  $\sigma \in \Sigma_{u,s}$ , where  $u$  is of length  $|u| = n$ , that  $F(\sigma(x_1^u, \dots, x_n^u)) = f(\sigma)$  where  $\sigma(x_1^u, \dots, x_n^u) : u \rightarrow s$  is the morphism in  $\mathbf{T}_\Sigma$  corresponding to the  $\Sigma$ -tree:*



The above theorem is, of course, an example of an adjunction.

We can interpret the  $\Sigma$ -flowcharts in any of the categories  $\mathbf{ITh}_S$  ( $S$ -sorted iteration theories),  $\mathbf{RTh}_S$  ( $S$ -sorted rational theories),  $\mathbf{CTh}_S$  ( $S$ -sorted  $\omega$ -continuous theories), or  $\mathbf{ETh}_S$  ( $S$ -sorted iterative theories). For example, we can view any iteration theory  $\mathbf{T} \in \mathbf{ITh}_S$  as a Lawvere algebraic theory in  $\mathbf{Th}_S$  by forgetting about  $\dagger$ , and define an interpretation as a theory morphism  $F : \mathbf{T}_\Sigma \rightarrow \mathbf{T}$  in  $\mathbf{Th}_S$ , then for any  $\Sigma$ -flowchart  $\langle \beta, \alpha \rangle$ , we have that  $\langle F(\beta), F(\alpha) \rangle$  is a flowchart in  $\mathbf{ITh}_S$  and can thus be solved in  $\mathbf{ITh}_S$ . However, because  $\mathbf{T}_\Sigma$  is freely generated by  $\Sigma$ , we can give the interpretation of  $\mathbf{T}_\Sigma$  in  $\mathbf{T}$  by just giving an  $(S^* \times S)$ -indexed family of mappings  $f = \langle f_{u,s} : \Sigma_{u,s} \rightarrow \mathbf{T}(u, s) \mid u \in S^*, s \in S \rangle$ . Indeed this is just what we did in the example in Section 4.

Given a category of  $S$ -sorted algebraic theories

$$\mathbf{KTh}_S \in \{\mathbf{ITh}_S, \mathbf{RTh}_S, \mathbf{CTh}_S, \mathbf{ETh}_S\}$$

and some class  $\mathcal{I}$  of interpretations of  $\mathbf{T}_\Sigma$ , we ask if there exists a theory  $\mathbf{T} \in |\mathbf{KTh}_S|$  and an interpretation  $f_{\mathcal{I}} : \Sigma(\mathbf{T}(u, s) \mid u \in S^*, s \in S)$  such that for all pairs  $\langle \mathcal{F}_1, \mathcal{F}_2 \rangle$  of  $\Sigma$ -flowcharts,  $\mathcal{F}_1$  and  $\mathcal{F}_2$  are equivalent under

every interpretation in  $\mathcal{I}$  if, and only if, they are equivalent under the interpretation  $F_{\mathcal{I}}$ . We call  $F_{\mathcal{I}}$  a *Herbrand interpretation* for  $\mathcal{I}$ .

## 7.1 Free theories

From the well-known theorem of universal algebra to the effect that freely generated algebras exist in every equationally defined variety of algebras, we get the following result:

**Proposition 7.1.1.** *For any  $S$ -sorted signature  $\Sigma$ , there exists a Herbrand interpretation,  $\mathbf{I}_{\Sigma}$ , for the class of all interpretations of  $\mathbf{T}_{\Sigma}$  in the category  $\mathbf{ITh}_S$  of  $S$ -sorted iteration theories.*

**Proof.** [Sketch] As noted earlier, the  $S$ -sorted iteration theories are just a particular variety of  $S^* \times S^*$ -sorted universal algebras. In consequence, for any  $S^* \times S^*$ -indexed family of sets  $G$  there exists an  $S$ -sorted iteration theory freely generated by  $G$ . While the signature  $\Sigma$  is an  $S^* \times S$ -indexed family of sets, it is easily seen that the desired free algebra is the one resulting from extending  $\Sigma$  to the  $S^* \times S^*$ -indexed family of sets  $\bar{\Sigma}$  where

$$\bar{\Sigma}_{u,v} = \begin{cases} \Sigma_{u,v} & \text{if } |v| = 1, \text{ i.e. if } v \in S \\ \emptyset & \text{if } |v| \neq 1. \end{cases}$$

■

Unfortunately the above result doesn't tell us very much about  $\mathbf{I}_{\Sigma}$  other than that it exists. Fortunately we can do much better, namely we can give an explicit construction for the free rational theory generated by  $\Sigma$ ,  $\mathbf{RT}_{\Sigma}$ , and Ésik [1980] has shown that, if we forget the ordering on  $\mathbf{RT}_{\Sigma}$ , then we get an iteration theory freely generated by  $\Sigma$ . Recall, from Definition 3.2.1, that  $\mathbf{CT}_{\Sigma}$  is the free  $\omega$ -continuous  $\Sigma$ -theory constructed from partial  $\Sigma$ -trees, and recall from Definition 3.2.4 that  $\mathbf{FT}_{\Sigma}$  is the sub-theory of  $\mathbf{CT}_{\Sigma}$  constructed from finite  $\Sigma$ -trees.

**Definition 7.1.2.** Let  $\Sigma$  be an  $S$ -sorted signature. We define  $\mathbf{RT}_{\Sigma}$ , the free  $S$ -sorted rational theory generated by  $\Sigma$  to be such that for each  $u, v \in S^*$ ,  $\mathbf{RT}_{\Sigma}(u, v)$  consists of all  $|v|$ -tuples of partial  $u$ -ary  $\Sigma$ -trees,  $\beta \bullet \alpha^{\nabla}$  in  $\mathbf{CT}_{\Sigma}$ , where  $\beta : w + u \rightarrow v$  and  $\alpha : w + u \rightarrow w$  in  $\mathbf{FT}_{\Sigma}$  for some  $w \in S^*$ , and where composition, tupling, identities and the distinguished morphisms are given by Theorem 6.0.1 (recall that every rational theory is an iteration theory).

Of course we have to show that  $\mathbf{RT}_{\Sigma}$  deserves to be called 'free':

**Theorem 7.1.3.** *(The rational closure theorem): The theory  $\mathbf{RT}_{\Sigma}$  is the rational theory freely generated by  $\Sigma$ , that is, if  $\mathbf{T}$  is any  $S$ -sorted rational theory and  $I = \langle I_{u,s} : \Sigma_{u,s} \rightarrow \mathbf{T}(u, s) \mid u \in S^*, s \in S \rangle$  is any  $S^* \times S$ -indexed family of maps, then there exists a unique rational theory morphism*



$\bar{I} : \mathbf{RT}_\Sigma \rightarrow \mathbf{T}$  that extends  $I$  in the sense that  $I(\sigma) = \bar{I}(\sigma(x_1^u, \dots, x_{|u|}^u))$  for all  $u \in S^*$ ,  $s \in S$ , and  $\sigma \in \Sigma_{u,s}$ .

**Proof.** [Sketch]

It can be shown that  $\mathbf{FT}_\Sigma$ , is the free ordered theory generated by  $\Sigma$  [Wagner *et al.*, 1976]. From that it follows that there is an ordered theory morphism  $\hat{I} : \mathbf{FT}_\Sigma \rightarrow \mathbf{T}$  extending  $I$ . We claim that the desired morphism  $\bar{I}$  is such that, for each  $\beta \bullet \alpha^\nabla$  in  $\mathbf{RT}_\Sigma$ ,

$$\bar{I}(\beta \bullet \alpha^\nabla) = \hat{I}(\beta) \bullet (\hat{I}(\alpha))^\nabla.$$

The only tricky part is showing that  $\bar{I}$  is well-defined. That is, that  $\beta_1 \bullet \alpha_1^\nabla = \beta_2 \bullet \alpha_2^\nabla$  implies  $\bar{I}(\beta_1 \bullet \alpha_1^\nabla) = \bar{I}(\beta_2 \bullet \alpha_2^\nabla)$ . This is done by first showing that the rational sequences for  $\beta_1 \bullet \alpha_1^\nabla$  and  $\beta_2 \bullet \alpha_2^\nabla$  in  $\mathbf{CT}_\Sigma$  must be cofinal in  $\mathbf{CT}_\Sigma$ . The fact that  $\hat{I}$  preserves the ordering then establishes the well-definedness of  $\bar{I}$ .

It remains to prove that  $\bar{I}$  is a rational theory morphism. As a first step we show that it extends  $\hat{I}$  and thus, in particular, that it preserves identity morphisms and distinguished morphisms. Let  $\gamma \in \mathbf{FT}_\Sigma(u, s)$ , then we have

$$\begin{aligned} \bar{I}(\gamma) &= \bar{I}(\gamma \bullet 0_u^\nabla) & (5.2.2.1) \\ &= \hat{I}(\gamma) \bullet (\hat{I}(0_u))^\nabla & \text{def. } \bar{I} \\ &= \hat{I}(\gamma) \bullet 0_u^\nabla & \hat{I} \text{ is a theory morphism} \\ &= \hat{I}(\gamma) \bullet 1_u & (5.2.2.1) \\ &= \hat{I}(\gamma). \end{aligned}$$

Using essentially the same argument steps, we can use (5.2.2.2) to show that  $\bar{I}$  preserves composition, (5.2.2.3) to show that  $\bar{I}$  preserves tupling, and (5.2.2.4) to show that  $\bar{I}$  preserves  $\nabla$ . This shows that  $\bar{I}$  is a rational theory morphism, and it is then straightforward to prove that  $\bar{I}$  is unique. ■

**Proposition 7.1.4.** *Let  $\langle \beta_1, \alpha_1 \rangle$  and  $\langle \beta_2, \alpha_2 \rangle$  be  $\Sigma$ -flowcharts. Then  $\langle \beta_1, \alpha_1 \rangle$  and  $\langle \beta_2, \alpha_2 \rangle$  are equivalent under every rational interpretation (i.e.,  $\bar{I}(\beta_1) \bullet (\bar{I}(\alpha_1))^\nabla = \bar{I}(\beta_2) \bullet (\bar{I}(\alpha_2))^\nabla$  for every  $I : \Sigma \rightarrow \mathbf{T}$ ,  $\mathbf{T}$  rational) iff they are equivalent in  $\mathbf{RT}_\Sigma$ .*

Clearly, if we forget the ordering on the hom-sets (carriers) of  $\mathbf{RT}_\Sigma$  then the result is an iteration theory.

**Theorem 7.1.5 (Ésik [1980]).** *The free rational theory  $\mathbf{RT}_\Sigma$ , with the order forgotten, is a free iteration theory generated by  $\Sigma$ . That is,  $\mathbf{I}_\Sigma \cong \mathbf{RT}_\Sigma$ .*

## 7.2 The general case

The free theories given above provide a Herbrand interpretation for the class of all interpretations of  $\Sigma$ -flowcharts. We now want to look at the more general case where the class  $\mathcal{I}$  of interpretations is a subclass of the class of all interpretations.

There are, at least, three contexts in which it is worthwhile looking at this question: iteration theories, rational theories, and  $\omega$ -continuous theories.

For iteration theories the answer is given by a straightforward application of universal algebra.

**Proposition 7.2.1.** *Let  $\Sigma$  be an  $S$ -sorted signature and let  $\mathcal{I}$  be a non-empty class of interpretations of  $\Sigma$ , then there exists an iteration theory  $\mathbf{T}_{\Sigma, \mathcal{I}}$  which is Herbrand for  $\mathcal{I}$ .*

**Proof.** Each interpretation  $I : \Sigma \rightarrow \mathbf{T}_I$  in  $\mathcal{I}$  gives us an iteration theory morphism  $\bar{I} : \mathbf{I}_\Sigma \rightarrow \mathbf{T}_I$ , which, in turn, induces a congruence  $Q_I$  on  $\mathbf{I}_\Sigma$ . Let  $Q_{\mathcal{I}} = \bigcap \{Q_I \mid I \in \mathcal{I}\}$ . Then the quotient algebra  $\mathbf{I}_\Sigma / Q_{\mathcal{I}}$  is the desired iteration theory. (To get more of an idea of the details see the following lemmas and the proof of (7.2.8) and read ‘congruence’ for ‘preorderance’.) ■

Essentially the same kind of argument works for rational theories, except that, since they are ordered, we can no longer work with congruences but we need a more general notion, namely the following:

**Definition 7.2.2.** Let  $\mathbf{T}$  be an  $S$ -sorted ordered algebraic theory. By a *preorderance on  $\mathbf{T}$*  we mean an  $(S^* \times S^*)$ -indexed family

$$Q = \langle Q_{u,v} \subseteq \mathbf{T}(u,v) \times \mathbf{T}(u,v) \mid u, v \in S^* \rangle$$

such that

(7.2.2.1) For all  $u, v \in S^*$ ,  $Q_{u,v}$  is a preorder on  $\mathbf{T}(u,v)$ , i.e. a reflexive, transitive relation.

(7.2.2.2) For all  $u, v \in S^*$ , if  $\alpha, \beta \in \mathbf{T}(u,v)$  and  $\alpha \sqsubseteq_{u,v} \beta$ , then  $\langle \alpha, \beta \rangle \in Q_{u,v}$ . Recall that  $\sqsubseteq_{u,v}$  is the ordering on  $\mathbf{T}(u,v)$  that comes with the ordered theory  $\mathbf{T}$ .

(7.2.2.3) If  $\langle \alpha_1, \alpha_2 \rangle \in Q_{u,v}$  and  $\langle \beta_1, \beta_2 \rangle \in Q_{v,w}$ , then  $\langle \beta_1 \bullet \alpha_1, \beta_2 \bullet \alpha_2 \rangle \in Q_{u,w}$ .

(7.2.2.4) If  $\langle \alpha_i, \beta_i \rangle \in Q_{u,v_i}$  for  $i = 1, \dots, |v|$ , then

$$\langle (\alpha_1, \dots, \alpha_{|v|}), (\beta_1, \dots, \beta_{|v|}) \rangle \in Q_{u,v}.$$

(7.2.2.5) If  $\mathbf{T}$  is a rational algebraic theory then we say that  $Q$  is a *rational preorderance* providing that it satisfies the following additional condition:

If  $\alpha : u + v \rightarrow u$ ,  $\gamma : u \rightarrow w$  and  $\tau : t \rightarrow v$ , and  $\beta : t \rightarrow w$  such that  $\langle \gamma \bullet \alpha^{(k)} \bullet \tau, \beta \rangle \in Q_{t,w}$ , then  $\langle \gamma \bullet \alpha^\dagger \bullet \tau, \beta \rangle \in Q_{t,w}$ .

Note that only the last condition relates directly to rationality.

Given a rational theory  $\mathbf{T}$  and a rational preorderance  $Q$  on  $\mathbf{T}$  let  $E[Q]$  be the largest  $(S^* \times S^*)$ -indexed family of equivalence relations contained in  $Q$ :

$$E[Q]_{u,v} = \{ \langle \alpha, \beta \rangle \mid \langle \alpha, \beta \rangle \in Q_{u,v} \text{ and } \langle \beta, \alpha \rangle \in Q_{u,v} \}.$$

For each  $u, v \in S^*$ , let  $\mathbf{T}/Q(u, v)$  denote the family of  $E[Q]$ -equivalence classes of  $\mathbf{T}(u, v)$ . Given  $\alpha \in \mathbf{T}(u, v)$  let  $[\alpha]$  denote its  $E[Q]$ -equivalence class. Given  $\alpha, \beta \in \mathbf{T}(u, v)$  define  $\sqsubseteq_Q$  to be the relation on  $\mathbf{T}/Q$  such that  $[\alpha] \sqsubseteq_Q [\beta]$  iff  $\langle \alpha, \beta \rangle \in Q$  (dropping subscripts for notational convenience).

**Lemma 7.2.3.** *Let  $\mathbf{T}$  be a rational theory and let  $Q$  be a rational pre-orderance on  $\mathbf{T}$ . Then  $\sqsubseteq_Q$  is a partial order, and  $\mathbf{T}/Q$  equipped with  $\sqsubseteq_Q$  is a rational theory with composition  $[\alpha] \bullet [\beta] = [\alpha \bullet \beta]$ , distinguished morphisms  $[x_i^u]$ , and tupling  $([\beta_1], \dots, [\beta_n]) = [(\beta_1, \dots, \beta_n)]$ .*

**Lemma 7.2.4.** *Let  $\mathbf{T}$  and  $\mathbf{T}'$  be  $S$ -sorted rational theories and let  $F : \mathbf{T} \rightarrow \mathbf{T}'$  be a rational theory morphism. For each  $u, v \in S^*$  let*

$$Q[F]_{u,v} = \{ \langle \alpha, \beta \rangle \mid \alpha, \beta \in \mathbf{T}(u, v) \text{ and } F(\alpha) \sqsubseteq_{u,v} F(\beta) \text{ in } \mathbf{T}' \},$$

*then  $Q[F]$  is a rational pre-orderance on  $\mathbf{T}$ .*

**Lemma 7.2.5.** *The canonical map  $F[Q] : \mathbf{T} \rightarrow \mathbf{T}/Q$  sending  $\alpha \in \mathbf{T}$  to its  $E[Q]$ -equivalence class in  $\mathbf{T}/Q$  is a rational theory morphism and  $Q$  is the induced rational pre-orderance of  $F[Q]$ , i.e.  $Q[F[Q]] = Q$ .*

**Lemma 7.2.6.** *Let  $\mathbf{T}$  be a rational theory. Then the set  $\mathcal{Q}[\mathbf{T}]$  of all rational pre-orderances on  $\mathbf{T}$  forms a complete lattice under the natural ordering:  $Q \subseteq Q'$  iff  $Q_{u,v} \subseteq Q'_{u,v}$  for all  $u, v \in S^*$ .*

**Lemma 7.2.7.** *Let  $G : \mathbf{T} \rightarrow \mathbf{T}'$  be a rational theory morphism, let  $Q$  be a rational pre-orderance on  $\mathbf{T}$  and let  $F[Q] : \mathbf{T} \rightarrow \mathbf{T}/Q$  be the rational theory morphism induced by  $Q$ . Let  $Q[G]$  be the rational pre-orderance induced by  $G$ . Then  $Q \subseteq Q[G]$  implies there exists a unique rational theory morphism  $I : \mathbf{T}/Q \rightarrow \mathbf{T}'$  such that  $G = I \bullet F[Q]$ .*

**Theorem 7.2.8.** *Given  $\Sigma$ , let  $\mathcal{I}$  be a class of interpretations for  $\Sigma$ . Then there exists a rational theory  $\mathbf{R}_{\mathcal{I}}$  and an interpretation  $H : \Sigma \rightarrow \mathbf{R}_{\mathcal{I}}$  such that for all  $\alpha, \beta \in \mathbf{RT}_{\Sigma}$   $\bar{H}(\alpha) = \bar{H}(\beta)$  iff  $\bar{I}(\alpha) = \bar{I}(\beta)$  for all  $I \in \mathcal{I}$ .*

**Proof.** For each  $I : \Sigma \rightarrow \mathbf{T}$  in  $\mathcal{I}$  let  $Q[\bar{I}]$  be the pre-orderance induced by  $\bar{I} : \mathbf{RT}_{\Sigma} \rightarrow \mathbf{T}$  (Theorem 7.1.2, Lemma 7.2.4). Let  $Q_{\mathcal{I}} = \bigcap \{ Q[\bar{I}] \mid I \in \mathcal{I} \}$ . Then  $Q_{\mathcal{I}}$  is a pre-orderance on  $\mathbf{RT}_{\Sigma}$  by Lemma 7.2.6. Now let  $\mathbf{R}_{\mathcal{I}} = \mathbf{RT}_{\Sigma}/Q_{\mathcal{I}}$  and let  $\bar{H} : \mathbf{RT}_{\Sigma} \rightarrow \mathbf{R}_{\mathcal{I}}$  be the theory morphism induced by  $Q_{\mathcal{I}}$ , Lemma 7.2.5. Then

$$\begin{aligned} \bar{H}(\alpha) &= \bar{H}(\beta) \\ \text{iff } \langle \alpha, \beta \rangle &\in E[Q_{\mathcal{I}}] \\ \text{iff } \langle \alpha, \beta \rangle &\in E[Q[\bar{I}]] \text{ for all } I \in \mathcal{I} \\ \text{iff } \bar{I}(\alpha) &= \bar{I}(\beta) \text{ for all } I \in \mathcal{I}. \end{aligned}$$

The desired interpretation  $H : \Sigma \rightarrow \mathbf{R}_{\mathcal{I}}$  is just  $\bar{H}$  restricted to  $\Sigma$ . ■

The class  $\mathcal{I}$  is usually a class of interpretations with some ‘property of interest’. Theorem 7.1.4 is used to produce a Herbrand interpretation for  $\mathcal{I}$

but there is no guarantee that  $H : \Sigma \rightarrow \mathbf{R}_I$  will be in  $\mathcal{I}$ . An important class of examples arises when the ‘property’ is described by means of defining relations. The idea is the same as that of specifying groups by means of ‘defining relations’ (there are usually equations or identities, but here inequations  $e_1 \sqsubseteq e_2$ ) and requiring that those relations be satisfied.

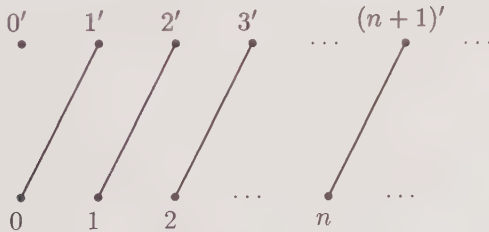
**Definition 7.2.9.** An *inequation* is a pair  $\langle e_1, e_2 \rangle \in \mathbf{RT}_\Sigma(u, s) \times \mathbf{RT}_\Sigma(u, s)$  for some  $u \in S^*$  and  $s \in S$ . An interpretation  $I : \Sigma \rightarrow \mathbf{T}$  *satisfies*  $\langle e_1, e_2 \rangle$  iff  $\bar{I}(e_1) \sqsubseteq_{u,s} \bar{I}(e_2)$ . If  $\mathcal{E}$  is a set of inequations then  $I$  satisfies  $\mathcal{E}$  iff  $I$  satisfies each  $\langle e_1, e_2 \rangle \in \mathcal{E}$ .

**Theorem 7.2.10.** For any set of inequations  $\mathcal{E}$ , there is a rational theory  $\mathbf{RT}_{\Sigma, \mathcal{E}}$  and an interpretation  $I_{\mathcal{E}} : \Sigma \rightarrow \mathbf{RT}_{\Sigma, \mathcal{E}}$  such that  $I_{\mathcal{E}}$  satisfies  $\mathcal{E}$  and if  $I : \Sigma \rightarrow \mathbf{T}$  is any interpretation satisfying  $\mathcal{E}$ , then there exists a unique morphism of rational theories  $H : \mathbf{RT}_{\Sigma, \mathcal{E}} \rightarrow \mathbf{T}$ , such that  $\bar{I} = H \bullet \bar{I}_{\mathcal{E}}$ .

This can be restated by saying that if  $I$  satisfies  $\mathcal{E}$  then  $\bar{I}$  factors uniquely through  $\bar{I}_{\mathcal{E}}$ : two  $\Sigma$ -flowcharts have the same behaviour under all interpretations satisfying  $\mathcal{E}$  (are  $\mathcal{E}$ -equivalent) iff they have identical behaviours under the interpretation  $I_{\mathcal{E}}$ .

The  $S$ -sorted  $\omega$ -continuous algebraic theories are an important subcategory of the  $S$ -sorted rational theories. This becomes particularly evident in the next section where we employ  $\omega$ -continuous theories to construct recursion hierarchies. In consequence, there is interest in looking at interpretations  $I : \Sigma \rightarrow \mathbf{T}$  in the category,  $\mathbf{CTh}_S$ , of  $S$ -sorted algebraic theories. Unfortunately  $\omega$ -continuous theories are not as nicely behaved as ordinary, iteration, and rational theories. For example, they are not closed under congruences.

**Example 7.2.11.** Let  $\mathbf{N} = \{0, 1, 2, \dots, n, \dots\}$  and  $\mathbf{N}' = \{0', 1', 2', \dots, n', \dots\}$  be two disjoint copies of the natural numbers. Let  $\Sigma$  be the one-sorted signature with  $\Sigma_0 = \mathbf{N} \cup \mathbf{N}'$  and  $\Sigma_p = \emptyset$  for all  $p > 0$ . Let  $Q$  be the preordering on  $\mathbf{T}_\Sigma$  generated by the inequations  $\langle n, (n+1)' \rangle$  for all  $n \in \mathbf{N}$ . Then  $\mathbf{T} = \mathbf{CT}_\Sigma / Q$  is an  $\omega$ -continuous theory where  $\mathbf{T}(0, 1)$  has the following Hasse diagram:



If we now let  $K$  be the congruence generated by the equalities  $n = n'$  then  $\mathbf{T}/K$  is an algebraic theory, but it is not  $\omega$ -continuous since it contains

the  $\omega$ -chain  $[0], [1], [2], \dots, [n], \dots$  and this  $\omega$ -chain does not have a least upper bound in  $\mathbf{T}/K$ .

The problem is that the familiar construction for quotient algebras works by forming equivalence classes of existing elements and thus does not create the new elements needed for  $\omega$ -continuity. There are two ways to get around this problem, one being to introduce an appropriate closure operation, and the other being to restrict our attention to preorderances that do not create new  $\omega$ -chains. Both approaches work, and have their place in the theory.

The key to the closure-operation approach is the following lemma:

**Lemma 7.2.12.** *Let  $\phi : \mathbf{T} \rightarrow \mathbf{T}'$  be a morphism of ordered theories (so  $\phi$  is strict and order preserving, see Definition 3.1.2) then there exists an  $\omega$ -continuous theory  $\bar{\mathbf{T}}$  and an  $\omega$ -continuous morphism of ordered theories  $u_\phi : \mathbf{T}' \rightarrow \bar{\mathbf{T}}$  such that*

*$u_\phi \bullet \phi$  is  $\omega$ -continuous.*

*If  $g : \mathbf{T}' \rightarrow \mathbf{T}''$  is any morphism of ordered theories such that  $g \bullet \phi$  is  $\omega$ -continuous, then there is a unique  $\omega$ -continuous theory morphism  $g^\sharp : \bar{\mathbf{T}} \rightarrow \mathbf{T}''$  such that  $g^\sharp \bullet u_\phi = g$ .*

$$\begin{array}{ccccc}
 \mathbf{T} & \xrightarrow{\phi} & \mathbf{T}' & \xrightarrow{u_\phi} & \bar{\mathbf{T}} \\
 & & & \searrow g & \downarrow g^\sharp \\
 & & & & \mathbf{T}''
 \end{array}$$

**Proof.** The proof can be found in [Bloom *et al.*, 1983]. The proof utilizes categorical concepts and is quite non-constructive. ■

**Theorem 7.2.13.** *By a plain preordering we mean one that is only required to satisfy the first four conditions of Definition 7.2.2. Let  $\mathbf{T}$  be an  $\omega$ -continuous theory, and let  $\mathcal{I}$  be a class of interpretations for  $\mathbf{T}$ . As above, in the proof of Theorem 7.2.8, let  $Q_{\mathcal{I}} = \bigcap \{Q[\bar{I}] \mid I \in \mathcal{I}\}$  (but with plain preorderances rather than rational preorderances). Then  $u_{F[Q_{\mathcal{I}}]} \bullet F[Q_{\mathcal{I}}]$  is a Herbrand interpretation for  $\mathcal{I}$ .*

The problem with the above solution is that it is non-constructive (non-effective) and does not, in general, lead to a complete proof system for deducing valid inequations  $t \sqsubseteq_{\mathcal{I}} t'$ . Clearly, such a proof system would be desirable for applications of these ideas to actual programming language constructs. Guessarian [Guessarian, 1975; Guessarian, 1981; Guessarian, 1985], and Guessarian and Courcelle [Courcelle and Guessarian, 1978] have successfully attacked this problem by investigating a number of restricted classes of interpretations that have proof systems and are



sufficiently broad to cover most computer science applications. Their work was done in a universal algebra framework rather than in the framework of Lawvere algebraic theories. The following is a very brief summary of part of their work, presented in an algebraic theoretic framework.

**Definition 7.2.14.** Let  $\Sigma$  be an  $S$ -sorted signature and let  $\mathbf{CT}_\Sigma$  be the  $\omega$ -continuous theory freely generated by  $\Sigma$ . Then a preordering  $Q$  on  $\mathbf{CT}_\Sigma$  will be said to be *inductive* if, for all  $\alpha, \beta \in \mathbf{CT}_\Sigma$ ,  $\langle \alpha, \beta \rangle \in Q$  iff there exist  $\bar{\alpha}, \bar{\beta} \in \mathbf{FT}_\Sigma$ , such that  $\bar{\alpha} \sqsubseteq \alpha$  and  $\bar{\beta} \sqsubseteq \beta$  in  $\mathbf{CT}_\Sigma$  and  $\langle \bar{\alpha}, \bar{\beta} \rangle \in Q$ .

Many authors, including Guessarian, use the term ‘algebraic’ in place of ‘inductive’—we use ‘inductive’ in order to avoid phrases such as ‘algebraic algebraic theory’.

A class of interpretations  $\mathcal{I}$  will be said to be *inductive* if the preordering  $Q_{\mathcal{I}} = \bigcap \{Q[\bar{I}] \mid I \in \mathcal{I}\}$  is inductive.

For any preordering  $Q$  on  $\mathbf{CT}_\Sigma$  define its *restriction*  $Q^f$  to  $\mathbf{FT}_\Sigma$  to be  $Q^f = Q \cap \mathbf{FT}_\Sigma^2$ . For any preordering  $Q$  on  $\mathbf{FT}_\Sigma$  define its *inductive closure*  $Q^a$  in  $\mathbf{CT}_\Sigma$  by  $\langle \alpha, \beta \rangle \in Q^a$  iff for each  $\gamma \in \mathbf{FT}_\Sigma$ ,  $\gamma \sqsubseteq \alpha$  in  $\mathbf{CT}_\Sigma$  implies there exists  $\gamma' \in \mathbf{FT}_\Sigma$ ,  $\gamma' \sqsubseteq \beta$  in  $\mathbf{CT}_\Sigma$ , such that  $\langle \gamma, \gamma' \rangle \in Q$ .

**Definition 7.2.15.** Let  $\mathbf{T}$  be an  $S$ -sorted ordered algebraic theory. A set  $J \subseteq \mathbf{T}(u, v)$  is said to be an *ideal* in  $\mathbf{T}(u, v)$  if it is downward closed with respect to  $\sqsubseteq_{u, v}$ .

Given an  $S$ -sorted algebraic theory  $\mathbf{T}$  define  $\mathbf{T}^\infty$ , the *ideal closure* of  $\mathbf{T}$ , to be the  $S$ -sorted theory where, for each  $u, v \in S^*$ ,  $\mathbf{T}^\infty(u, v)$  is the set of all ideals in  $\mathbf{T}(u, v)$ , and with composition operator  $\bullet_\infty$ , where, for  $\alpha \in \mathbf{T}^\infty(u, v)$  and  $\beta \in \mathbf{T}^\infty(v, w)$ ,

$$\beta \bullet_\infty \alpha = \{\gamma \in \mathbf{T}(u, v) \mid \exists \gamma_1 \in \alpha, \gamma_2 \in \beta, \gamma \sqsubseteq_{u, v} \gamma_2 \bullet \gamma_1\},$$

with  $\mathbf{T}^\infty(u, v)$  ordered by the set-theoretic inclusion on the ideals.

The inductive closure is weaker than the completion given in Theorem 7.2.12, it satisfies the following lemma.

**Lemma 7.2.16.** Let  $\phi : \mathbf{T} \rightarrow \mathbf{T}'$  be a morphism of ordered theories (so  $\phi$  is strict and order preserving, see Definition 3.1.2) then there exists a morphism of ordered theories  $u_\phi : \mathbf{T}' \rightarrow \mathbf{T}^\infty$  such that

If  $g : \mathbf{T}' \rightarrow \mathbf{T}''$  is any morphism of ordered theories then there is a unique  $\omega$ -continuous theory morphism  $g^\sharp : \mathbf{T}^\infty \rightarrow \mathbf{T}''$  such that  $g^\sharp \bullet u_\phi = g$ .

Then we have

**Theorem 7.2.17.** [Guessarian, 1981], A class of interpretations,  $\mathcal{I}$ , is inductive iff it satisfies one of the following equivalent conditions:

1.  $(Q_{\mathcal{I}}^f)^a = Q_{\mathcal{I}}$
2.  $(\mathbf{CT}_\Sigma / Q_{\mathcal{I}}^f)^\infty$  is Herbrand for  $\mathcal{I}$ .
3. There exists an inductive  $\mathcal{I}$ -interpretation.

## 8 Recursive hierarchies

In this section we give a presentation of ‘the M-construction’ of ADJ (see [Bloom *et al.*, 1983]), which provides a functorial construction of hierarchies of theories, of higher and higher type. The underlying construction is a special case of the Kleisli category construction from category theory, see [MacLane, 1971]. Several well-known hierarchies fit into this framework:

regular sets, context-free sets, indexed sets, ...

flowcharts, monadic recursion schemes, ...

elementary arithmetic, recursion schemes, recursion operators, ...

For example, this construction takes  $\mathbf{CT}_\Sigma$ , the free theory of partial  $\Sigma$ -trees, to a theory  $M(\mathbf{CT}_\Sigma)$  which contains  $\mathbf{Rec}_\Sigma$ , the free  $\omega$ -continuous  $\Sigma$ -recursion theory (see Definition 3.2.7).

The underlying idea here is quite independent of the concept of iteration operator, and, indeed, the real problem is to show that we can get the construction to cooperate with iteration.

**Definition 8.0.1.** We say that a category,  $\mathbf{Kth}_S$ , of algebraic theories is *M-able* if

- For each  $S$ -sorted signature  $\Sigma$  there is a theory  $\mathbf{T}_\Sigma$  in  $\mathbf{KTh}_S$  freely generated by  $\Sigma$ .

Recall that this means that there is a signature morphism  $\eta_\Sigma : \Sigma \rightarrow \langle \mathbf{T}_\Sigma(u, s) \mid u \in S^*, s \in S \rangle$  such that for any theory  $\mathbf{T}'$  in  $\mathbf{KTh}_S$ , and any signature morphism  $f : \Sigma \rightarrow \langle \mathbf{T}_\Sigma(u, s) \mid u \in S^*, s \in S \rangle$  there exists a unique theory morphism  $f^\sharp : \mathbf{T}_\Sigma \rightarrow \mathbf{T}'$  in  $\mathbf{KTh}_S$  such that, for every  $\sigma \in \Sigma_{u,s}$   $F_{u,s}(\eta_\Sigma(\sigma)) = f(\sigma)$ .

- If  $\mathbf{T}, \mathbf{T}' \in |\mathbf{KTh}_S|$  then they have a coproduct  $\mathbf{T} + \mathbf{T}'$  with injections  $\lambda_{\mathbf{T}, \mathbf{T}'} : \mathbf{T} \rightarrow \mathbf{T} + \mathbf{T}'$  and  $\kappa_{\mathbf{T}, \mathbf{T}'} : \mathbf{T}' \rightarrow \mathbf{T} + \mathbf{T}'$ .

Recall that this means that for any theory  $\mathbf{T}''$  and morphisms  $F : \mathbf{T} \rightarrow \mathbf{T}''$  and  $F' : \mathbf{T}' \rightarrow \mathbf{T}''$  in  $\mathbf{KTh}_S$  there exists a unique morphism  $[F, F'] : (\mathbf{T} + \mathbf{T}') \rightarrow \mathbf{T}''$  in  $\mathbf{KTh}_S$  such that  $[F, F'] \bullet \lambda_{\mathbf{T}, \mathbf{T}'} = F$  and  $[F, F'] \bullet \kappa_{\mathbf{T}, \mathbf{T}'} = F'$ .

**Definition 8.0.2.** Let  $\mathbf{KTh}_S$  be an *M-able* category of  $S$ -sorted algebraic theories. The ‘M-construction’ on  $\mathbf{KTh}_S$  is then defined as follows.

Let  $S^\sharp = (S^* \times S)^*$ , the set of strings on  $(S^* \times S)$ . Given  $w \in S^\sharp$ , let  $\Sigma^w$  denote the  $S$ -sorted signature where, for each  $u \in S^*$  and  $s \in S$ ,

$$\Sigma_{u,s}^w = \{f_i^w \mid w_i = \langle u, s \rangle\}.$$

Given  $\mathbf{T} \in \mathbf{KTh}_S$ , we construct an  $S^\sharp = (S^* \times S)$ -sorted algebraic theory  $M(\mathbf{T}) \in |\mathbf{Th}_{S^* \times S}|$  as follows:

- For  $v, w \in S^\sharp$  a morphism  $\alpha$  in  $M(\mathbf{T})(v, w)$  is a triple  $\langle v, \bar{\alpha}, w \rangle$  where  $\bar{\alpha}$  is theory morphism

$$\bar{\alpha} : \mathbf{T}_{\Sigma^w} \rightarrow (\mathbf{T} + \mathbf{T}_{\Sigma^v})$$

in  $\mathbf{KTh}_S$ . Because of the freeness of  $\mathbf{T}_{\Sigma^u}$ ,  $\bar{\alpha}$  is completely determined by its restriction to the signature morphism

$$\begin{aligned} \Sigma^w &\rightarrow \langle (\mathbf{T} + \mathbf{T}_{\Sigma^v})(u, s) \mid (u, s) \in S^* \times S \rangle \\ f_i^w &\mapsto \bar{\alpha}(f_i^w). \end{aligned}$$

Note that, if  $f_i^w \in \Sigma_{u,s}^w$  then  $\bar{\alpha}(f_i^w) \in (\mathbf{T} + \mathbf{T}_{\Sigma^v})(u, s)$ .

- Given morphisms  $\alpha : u \rightarrow v$  and  $\beta : v \rightarrow w$  in  $M(\mathbf{T})$ , their composite,  $\beta \bullet \alpha$ , is given by the theory morphism

$$\overline{\beta \bullet \alpha} = [\lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^u}}, \bar{\alpha}] \bullet \bar{\beta}$$

in  $\mathbf{KTh}_S$ , where, as shown in the diagram below,  $[\lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^u}}, \bar{\alpha}]$  is the unique morphism given by the coproduct such that

$$[\lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^u}}, \bar{\alpha}] \bullet \lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^v}} = \lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^v}}$$

and

$$[\lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^u}}, \bar{\alpha}] \bullet \kappa_{\mathbf{T}, \mathbf{T}_{\Sigma^v}} = \bar{\alpha}.$$

$$\begin{array}{ccccc} & & \mathbf{T} & & \\ & & \downarrow \lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^v}} & \searrow \lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^u}} & \\ & & & & \mathbf{T} + \mathbf{T}_{\Sigma^u} \\ \mathbf{T}_{\Sigma^w} & \xrightarrow{\bar{\beta}} & \mathbf{T} + \mathbf{T}_{\Sigma^v} & \xrightarrow{[\lambda_{\mathbf{T}, \mathbf{T}_{\Sigma^u}}, \bar{\alpha}]} & \mathbf{T} + \mathbf{T}_{\Sigma^u} \\ & & \uparrow \kappa_{\mathbf{T}, \mathbf{T}_{\Sigma^v}} & \nearrow \bar{\alpha} & \\ & & \mathbf{T}_{\Sigma^v} & & \end{array}$$

- Define the needed tupling operators to be such that given  $v, w \in (S^* \times S)^*$  and, for each  $i = 1, \dots, |w|$ , a morphism  $\alpha_i : v \rightarrow w_i$  in  $M(\mathbf{T})$  then their tupling,  $\alpha = (\alpha_1, \dots, \alpha_{|w|}) : v \rightarrow w$ , is determined by the morphism  $\bar{\alpha} : \mathbf{T}_{\Sigma^w} \rightarrow (\mathbf{T} + \mathbf{T}_{\Sigma^v})$  corresponding to the signature morphism

$$\begin{aligned} \Sigma^w &\rightarrow \langle (\mathbf{T} + \mathbf{T}_{\Sigma^v})(u, s) \mid (u, s) \in S^* \times S \rangle \\ f_i^w &\mapsto \alpha_i(f_1^{w_i}). \end{aligned}$$

- For each  $w \in (S^* \times S)^*$ , and  $i = 1, \dots, |w|$ , define the distinguished morphism  $x_i^w : w \rightarrow w_i$  in  $M(\mathbf{T})$  to be such that  $\overline{x_i^w} = \kappa_{\mathbf{T}, \mathbf{T}_{\Sigma^w}} \bullet g_i^\sharp$  where  $g_i$  is the signature morphism

$$g_i : \Sigma^{w_i} \rightarrow \Sigma^w$$

$$f_1^{w_i} \mapsto f_i^w.$$

After showing that these operations satisfy the definition of an algebraic theory, and that the construction is functorial, we obtain:

**Theorem 8.0.3.** *If  $\mathbf{KTh}_S$  is an  $M$ -able category of  $S$ -sorted theories then for each  $\mathbf{T} \in |\mathbf{KTh}_S|$ ,  $M(\mathbf{T})$  is an  $(S^* \times S)$ -sorted algebraic theory. Indeed, this construction gives us a functor*

$$M : \mathbf{KTh}_S \rightarrow \mathbf{Th}_{S^* \times S}.$$

**Proof.** [Sketch] We leave it to the reader to show that the above definitions satisfy the requirements for  $M(\mathbf{T})$  to be an  $(S^* \times S)$ -sorted theory. To show that  $M$  is a functor we need to define its behaviour on morphisms. Let  $F : \mathbf{T} \rightarrow \mathbf{T}'$  in  $\mathbf{KTh}_S$ , then  $M(F) : M(\mathbf{T}) \rightarrow M(\mathbf{T}')$  such that

- For every  $u \in (S^* \times S)^*$ ,  $(M(F))(u) = u$ .
- Given all  $u, v \in (S^* \times S)^*$  and  $\alpha \in (M(\mathbf{T}))(u, v)$  recall that  $\alpha$  is specified by a morphism  $\alpha : \mathbf{T}_{\Sigma^v} \rightarrow (\mathbf{T} + \mathbf{T}_{\Sigma^u})$  in  $\mathbf{KTh}_S$ . Define  $(M(F))(\alpha)$  to be the morphism in  $M(\mathbf{T}')(u, v)$  specified by  $(F + 1_{\mathbf{T}_{\Sigma^u}}) \bullet \alpha$

$$\begin{array}{ccc} \mathbf{T}_{\Sigma^v} & \xrightarrow{\alpha} & \mathbf{T} + \mathbf{T}_{\Sigma^u} \\ & \searrow (M(F))(\alpha) & \downarrow F + 1_{\mathbf{T}_{\Sigma^u}} \\ & & \mathbf{T} + \mathbf{T}'_{\Sigma^u} \end{array}$$

We leave the details of the proof to the reader. ■

**Example 8.0.4.** It is relatively straightforward to show that  $\mathbf{Rec}_\Sigma$ , the free  $\omega$ -continuous  $\Sigma$ -recursion theory of Example 3.2.5, is a sub-theory of  $M(\mathbf{CT}_\Sigma)$ . Here the relevant category,  $\mathbf{KTh}_S$  is the category of 1-sorted  $\omega$ -continuous theories,  $\mathbf{CTh}_S$  for  $S = \{s\}$ , a singleton set. We will show later that  $\mathbf{CTh}_S$  is  $M$ -able ; this example can be treated as a special case that does not require that general result.

- In that example,  $\Sigma$  is a 1-sorted theory,  $S = \{s\}$ , and so, for notational convenience, the elements of  $S^*$  are identified with the set of natural numbers,  $\omega$  (we write  $n$  for  $s^n$ ), and the elements of  $(S^* \times S)^*$  are identified with  $\omega^*$ . Keeping this in mind, it is easy to see that for any  $u \in (S^* \times S)^* = \omega^*$  that the signature  $F^u$  in (3.2.5) is the same as the signature  $\Sigma^u$  of Definition 8.0.2.
- Since we are dealing with  $\omega$ -continuous theories we see that for each  $u \in \omega^*$  that  $\mathbf{T}_{\Sigma^u} = \mathbf{CT}_{\Sigma^u}$ . But, as is easily shown,  $\mathbf{CT}_\Sigma + \mathbf{CT}_{\Sigma^u} =$

$\mathbf{CT}_{\Sigma+\Sigma^u}$ . Thus the morphisms in  $(\mathbf{CT}_{\Sigma} + \mathbf{CT}_{\Sigma^u})$  are exactly the partial  $(\Sigma + F'^u)$ -trees used in (3.2.5).

- Now a morphism in  $\mathbf{Rec}_{\Sigma}(u, v)$  for  $v \in \omega^n$ , was defined as an  $n$ -tuple  $(t_1, \dots, t_n)$  of partial  $(\Sigma + F^u)$ -trees where each  $t_i$  is of rec-sort  $v_i$ , while a morphism in  $M(\mathbf{CT}_{\Sigma})(v, u)$  is defined by a morphism  $\tau : \mathbf{CT}_{\Sigma, v} \rightarrow (\mathbf{CT}_{\Sigma} + \mathbf{CT}_{\Sigma^u})$  which, in turn, is determined by a signature morphism  $\hat{\tau} : \Sigma^v \rightarrow \langle (\mathbf{CT}_{\Sigma} + \mathbf{CT}_{\Sigma^u})(w, 1) \mid w \in \omega \rangle$ . But the condition for  $\hat{\tau}$  to be a signature morphism is exactly that it ‘preserve rec-sort’, i.e., that, for each it take elements of  $\Sigma_n^v$  to elements of  $(\mathbf{CT}_{\Sigma} + \mathbf{CT}_{\Sigma^u})(n, 1)$  for each  $n \in \omega$ . Thus the morphisms are identical in the two theories.
- It remains to show that the composition operations are the same in the two theories. The definition of composition given in (3.2.5) is a special case of that given above. It exploits the fact that  $\mathbf{CT}_{\Sigma} + \mathbf{CT}_{\Sigma^u}$  is free in  $\Sigma + \Sigma^u$  to produce an inductive definition.

The above theorem only asserts that  $M(\mathbf{T}) \in |\mathbf{Th}_{(S^* \times S)}|$  and, in particular, it does not assert that  $M(\mathbf{T})$  is an iteration theory. However, in order to able to define solutions for the ‘recursion equations’ in  $M(\mathbf{T})$  we need it to be an iteration theory.

Close inspection of the  $M$ -construction shows that it does not generally preserve the property of being iteration theory. Even if we restrict  $\mathbf{T}$  to being a rational theory it does not follow that  $M(\mathbf{T})$  is an iteration theory. Fortunately the  $M$ -construction works very nicely for  $\omega$ -continuous theories. That is,  $\mathbf{CTh}_S$  is  $M$ -able for all choices of  $S$ , and, if we take  $\mathbf{T} \in \mathbf{CTh}_S$  then  $M(\mathbf{T}) \in \mathbf{CTh}_{S^* \times S}$  so that iteration is well defined. Furthermore,  $M(\mathbf{T}) \in \mathbf{CTh}_{S^* \times S}$  means that we can repeat the process and define  $M(M(\mathbf{T}))$  and, more generally,  $M^n(\mathbf{T})$  for all  $n \in \omega$ .

The basic results needed to show that the  $M$ -construction can be carried out within  $\mathbf{CTh}_S$  are as follows:

**Proposition 8.0.5.** *Let  $\Sigma$  be an  $S$ -sorted signature, then  $\mathbf{CT}_{\Sigma}$ , as defined in Definition 3.2.3, is the free  $\omega$ -continuous theory generated by  $\Sigma$ .*

**Proposition 8.0.6.** *There is a functorial construction from  $\mathbf{Th}_S$  into  $\mathbf{CTh}_S$  which takes an ordinary theory  $\mathbf{T}$  to an  $\omega$ -continuous theory freely generated by  $\mathbf{T}$ .*

**Proof.** Take  $\phi = 1_{\mathbf{T}}$  in Lemma 7.2.12. Then  $\bar{\mathbf{T}}$  is the  $\omega$ -continuous theory freely generated by  $\mathbf{T}$ . This gives us a functor  $F_{con} : \mathbf{Oth}_S \rightarrow \mathbf{CTh}_S$  that is left adjoint to the inclusion functor  $U_{con} : \mathbf{CTh}_S \rightarrow \mathbf{Oth}_S$ . The desired functor is the composition of  $F_{con}$  with the functor  $F_{ord}$  of Proposition 3.1.3. ■

**Proposition 8.0.7.** *The category  $\mathbf{CTh}_S$  has coproducts.*



**Proof.** Again the proof will be found in [Bloom *et al.*, 1983]. It makes considerable use of Lemma 7.2.12. ■

**Corollary 8.0.8.** *The category  $\mathbf{CTh}_S$  is  $M$ -able.*

**Theorem 8.0.9.** *If  $\mathbf{T}$  is an  $S$ -sorted  $\omega$ -continuous algebraic theory with ordering  $\sqsubseteq_{u,v}$  on  $\mathbf{T}(u,v)$  for all  $u,v \in S^*$ , then  $M(\mathbf{T})$  is an  $(S^* \times S)$ -sorted  $\omega$ -continuous algebraic theory with ordering  $\sqsubseteq_{w,t}^M$  on  $M(\mathbf{T})(w,t)$  for all  $w,t \in (S^* \times S)^*$ , where for  $\alpha, \beta \in M(\mathbf{T})(w,t)$  with  $\bar{\alpha} : \mathbf{CT}_{\Sigma^t} \rightarrow \mathbf{T} + \mathbf{CT}_{\Sigma^w}$  and  $\bar{\beta} : \mathbf{CT}_{\Sigma^t} \rightarrow \mathbf{T} + \mathbf{CT}_{\Sigma^w}$ ,  $\alpha \sqsubseteq_{w,t} \beta$  iff for every  $u \in S^*$ ,  $s \in S$  and  $\sigma \in \Sigma_{u,s}^t$ ,  $\bar{\alpha}(\sigma) \sqsubseteq_{u,s} \bar{\beta}(\sigma)$ .*

**Theorem 8.0.10.** *Let  $\Sigma$  be an  $S$ -sorted signature, then  $M(\mathbf{CT}_{\Sigma}) \cong \mathbf{Rec}_{\Sigma}$  ( $\mathbf{Rec}_{\Sigma}$  as in Example 3.2.5).*

We can, of course, go further and construct  $M^n(\mathbf{CT})_{\Sigma}$  for all  $n \in \omega$ . Let us end by looking at two examples of ‘flowcharts’ in the  $M(M(\mathbf{CT}_{\Sigma})) = M(\mathbf{Rec}_{\Sigma})$  for  $\Sigma$  the one-sorted signature given in Example 3.2.7. Note that for  $\Sigma$  1-sorted we get that  $M(\mathbf{CT}_{\Sigma})$  is  $\omega$ -sorted and  $M(M(\mathbf{CT}_{\Sigma}))$  is  $(\omega^* \times \omega)$ -sorted.

We will show that the basic recursion operators: composition, minimalization, and primitive recursion, are definable in  $M(M(\mathbf{CT}_{\Sigma})) = M(\mathbf{Rec}_{\Sigma})$ . We write the morphisms as equations (as we did for the recursion equations in Example 3.2.7). Here ‘ $\bullet_1$ ’ denotes the composition in  $\mathbf{CT}_{\Sigma}$ , and ‘ $\bullet_2$ ’ denotes the composition in  $M(\mathbf{CT}_{\Sigma})$ . The lower-case  $f$ ’s are function variables and the  $F$ ’s are variables over functions on functions. In particular, but speaking informally,  $f_{n,i}$  denotes the  $i$ th function variable ranging over functions of  $n$  arguments, and, for example,  $F_{(n,p,q,r),j}$  is the  $j$ th function on functions having three arguments, the first of arity  $n$ , the second of arity  $p$ , the third of arity  $q$ , and returning a function of arity  $r$ .

The simplest is composition, there we have, for each choice of  $n, p \in \omega$  a composition operator defined as

$$\begin{aligned} F_1^{(n \cdot p \cdots p, p)}(f_1^{n \cdot p \cdots p, p}, f_2^{n \cdot p \cdots p, p}, \dots, f_{n+1}^{n \cdot p \cdots p, p}) \\ = f_1^{n \cdot p \cdots p, p} \bullet_1 (f_2^{n \cdot p \cdots p, p}, \dots, f_{n+1}^{n \cdot p \cdots p, p}). \end{aligned}$$

Then we have a ‘minimalization operator’ for each  $n > 0$ . The aim is to define a ‘function on functions’  $F_1^{n+1, n}$  such that applying  $F_1^{n+1, n}$  to a function  $f$  of  $n+1$  arguments will yield a function  $g = F_1^{N+1, f}(f)$  of  $n$  arguments, such that for any arguments  $y_1, \dots, y_n$  we will have  $g(y_1, \dots, y_n)$  is the smallest natural number  $y$  such that  $f(y, y_1, \dots, y_n) = 0$ . We claim that the desired definition is

$$\begin{aligned} F_1^{(n+1, n)}(f_1^{(n+1)}) \\ = If(x_1^{n+1}, 0, 0, \\ \text{succ}(F_1^{(n+1, n)}(f_1^{n+1}(\text{succ}(x_1^{n+1}), x_2^{n+1}, \dots, x_{n+1}^{n+1}))))). \end{aligned}$$

Finally we have a primitive recursion operator for each  $n$ . The aim is to define a 'function on functions',  $F_1^{(n \cdot n+2, n+1)}$ , such that given any  $n$  argument function  $f_1$  and any  $n+2$  argument function  $f_2$  that  $g = F_1^{(n \cdot n+2, n+1)}(f_1, f_2)$  is an  $n+1$  argument function such that for any arguments  $y, x_1, \dots, x_n \in \omega$ :

$$g(y, x_1, \dots, x_n) = \begin{cases} f_1(x_1, \dots, x_n) & \text{if } y = 0 \\ f_2(z, f_1(z, x_1, \dots, x_n), x_1, \dots, x_n) & \text{if } y = \text{succ}(z) \end{cases}$$

We claim the desired  $M(M(\mathbf{CT}_\Sigma))$ -morphism is

$$\begin{aligned} & F_1^{(n \cdot n+2, n+1)}(f_1^{n \cdot n+2}, f_2^{n \cdot n+2}) \\ &= If(x_1^{n+1}, 0, f_1^{n \cdot n+2} \bullet_1 (x_2^{n+1}, \dots, x_{n+1}^{n+1}), \\ & \quad f_2^{n \cdot n+2}(\text{pred}(x_1^{n+1}), \\ & \quad F_1^{(n \cdot n+2, n+1)}(f_1^{n \cdot n+1}, f_2^{n \cdot n+2}(\text{pred}(x_1), x_2, \dots, x_n)), \\ & \quad x_1, \dots, x_{n+1})). \end{aligned}$$

## References

- [Bloom and Tindell, 1980] S. L. Bloom and R. Tindell. Compatible orderings on the metric theory of trees. *SIAM J. Computing*, 9(4):683-691, 1980.
- [Bloom and Wagner, 1985] S. L. Bloom and E. G. Wagner. Many-sorted theories and their algebras with some applications to data types. In M. Nivat and J. C. Reynolds, editors, *Algebraic methods in semantics*, pages 133-168. Cambridge University Press, Cambridge, 1985.
- [Bloom et al., 1980a] S. L. Bloom, C. C. Elgot, and J. B. Wright. Solutions of the iteration equation and extensions of the scalar iteration operation. *SIAM J. Computing*, 9(1):25-45, 1980.
- [Bloom et al., 1980b] S. L. Bloom, C. C. Elgot, and J. B. Wright. Vector iteration in pointed iterative theories. *SIAM J. Computing*, 9(3):525-540, 1980.
- [Bloom et al., 1983] S. L. Bloom, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Recursion and iteration in continuous theories: The m-construction. *Journal of Computer and System Sciences*, 27(2):148-164, 1983.
- [Courcelle and Guessarian, 1978] B. Courcelle and I. Guessarian. On some classes of interpretations. *Journal of Computer and System Sciences*, 17:388-413, 1978.
- [Eilenberg and Wright, 1967] S. Eilenberg and J. B. Wright. Automata in general algebras. *Information and Control*, 11:452-470, 1967.
- [Elgot, 1975] C. C. Elgot. Monadic computation and iterative algebraic theories. In H. E. Rose and J. S. Shepherdson, editors, *Proceedings of the*

- Logic Colloquium '73, Studies in Logic*, pages 175–230. North Holland, Amsterdam, 1975.
- [Ésik, 1980] Z. Ésik. Identities in iterative and rational algebraic theories. *Computational Linguistics and Computer Languages*, 14(7):183–207, 1980.
- [Ésik, 1983] Z. Ésik. Algebras of iteration theories. *Journal of Computer and System Sciences*, 27:291–303, 1983.
- [Goguen and Burstall, 1977] J.A. Goguen and R. M. Burstall. Putting theories together to make specifications. In *Proc. 5th Intl. Joint Conf. on Artificial Intelligence, Cambridge, Mass.*, pages 1045–1058, 1977.
- [Goguen and Burstall, 1981] J.A. Goguen and R. M. Burstall. The semantics of clear, a specification language. In *Proc. of Advanced Course on Abstract Software Specifications, Copenhagen*, pages 292–332. Springer LNCS 86, 1981.
- [Goguen et al., 1978] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, IV, Data Structuring*, pages 80–149. Prentice-Hall, 1978.
- [Gordon, 1979] M. J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.
- [Guessarian, 1975] I. Guessarian. *Schemas récursifs polyadiques: équivalences et classes d'interprétation*. PhD thesis, Univ. Paris, 1975.
- [Guessarian, 1981] I. Guessarian. *Algebraic Semantics*, volume LNCS 99. Springer-Verlag, Berlin, 1981.
- [Guessarian, 1985] I. Guessarian. Survey on classes of interpretations and some of their applications. In M. Nivat and J. C. Reynolds, editors, *Algebraic methods in semantics*, pages 383–410. Cambridge University Ppress, Cambridge, 1985.
- [Lawvere, 1963] W. Lawvere. Functorial semantics of algebraic theories. *Proc. Nat. Acad. Sci.*, 21(1):1–23, 1963.
- [MacLane, 1971] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971.
- [Mezei and Wright, 1967] J. Mezei and J. B. Wright. Algebraic automata and context-free sets. *Information and Control*, 11(1-2):3–29, 1967.
- [Ștefănescu, 1987] Gh. Ștefănescu. On flowchart theories: Part i. the deterministic case. *Journal of Computer and System Science*, 35:163–191, 1987.
- [Thatcher et al., 1978] J. W. Thatcher, E. G. Wagner, and J. B. Wright. A uniform approach to inductive posets and inductive closure. *Theoretical Computer Science*, 7:57–77, 1978.

- [Thatcher *et al.*, 1981] J. W. Thatcher, E. G. Wagner, and J. B. Wright. More advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.
- [Wagner and Ehrig, 1987] E. G. Wagner and H. Ehrig. Canonical constraints for parameterized data types. *Theoretical Computer Science*, 50:323–349, 1987.
- [Wagner *et al.*, 1976] E. G. Wagner, J. W. Thatcher, J. W. Wright, and J. A. Goguen. Some fundamentals of order-algebraic semantics. In *Mathematical Foundations of Computer Science 1976, LNCS 45*, pages 153–168. Springer-Verlag, 1976.
- [Wright *et al.*, 1976] J. B. Wright, J. W. Thatcher, E. G. Wagner, and J. A. Goguen. Rational algebraic theories and fixed-point solutions. In *Proceedings 17th IEEE Symposium on Foundations of Computing, Houston, Texas*, pages 147–158, 1976.





# The Semantics of Types in Programming Languages

Carl A. Gunter

---

## Contents

1	Introduction . . . . .	396
2	Types in programming . . . . .	397
	2.1 Higher types . . . . .	397
	2.2 Recursive types . . . . .	400
	2.3 Parametric polymorphism . . . . .	402
	2.4 Subtypes . . . . .	404
3	Simple types as sets . . . . .	408
	3.1 Types and equations . . . . .	409
	3.2 Sets as a model . . . . .	412
	3.3 Type frames . . . . .	415
	3.4 Completeness for sets . . . . .	418
4	Simple types as domains . . . . .	420
	4.1 A programming language for computable functions . . . . .	420
	4.2 Operational semantics . . . . .	422
	4.3 Operational equivalence . . . . .	425
	4.4 bc-domains and dI-domains . . . . .	426
	4.5 Full abstraction . . . . .	427
5	Types as invariants . . . . .	430
	5.1 Run-time safety . . . . .	430
	5.2 Implicit types . . . . .	434
	5.3 Run-time safety for assignments and continuations. . . . .	441
6	Types as subsets . . . . .	446
	6.1 Untyped $\lambda$ -calculus . . . . .	446
	6.2 What is a model of the untyped $\lambda$ -calculus? . . . . .	448
	6.3 What models of the untyped $\lambda$ -calculus are there? . . . . .	449
	6.4 Inclusive subsets as types . . . . .	451
	6.5 Subtyping as subset inclusion . . . . .	455
7	Types as partial equivalence relations . . . . .	458
	7.1 Sets as a model of $ML_0$ types . . . . .	458

7.2	Another typing system for $ML_0$ . . . . .	460
7.3	The polymorphic $\lambda$ -calculus . . . . .	461
7.4	Sets as a model of polymorphic types? . . . . .	464
7.5	Simple types as PERs . . . . .	466
7.6	PERs as a model of polymorphic types . . . . .	468
8	Conclusion . . . . .	470

1 Introduction

In the twentieth century, there have been at least two lines of development of the notion of a *type*. One of these uses types to conquer problems in the foundations of mathematics. For example, type distinctions can resolve troubling paradoxes that lead to inconsistent systems. But a second, more recent, line of investigation into types pursues their application in programming languages. Although computer architectures themselves suggest few type distinctions, ‘higher-level’ programming languages generally use a classification of data into types to serve a variety of different purposes. There are at least four motives for doing this.

One of the earliest reasons for using types in programming languages such as Fortran was the enhancement of efficiency. For example, if a variable is declared to be an array of integers having a given number of entries, then it is possible to provide good space management for the variable in computer memory. A programmer’s indication of the type of a datum might also save pointless testing in a running program. This use of types has remained a basic motivation for their presence in many modern languages and will remain an important application.

A second motivation for the use of types was appreciated by the end of the 1960s. This was their role as a programming *discipline*. Types could be used to enforce restrictions on the shape of well-formed programs. One key benefit in doing this was the possibility of detecting flaws in programs in the form of compile-time type errors. If a mistake in a program can be detected at the point that the program is submitted for compilation into machine code, then this mistake can be corrected promptly rather than being discovered later in a run of the program on data. Although the software engineering gains obtained in this way are widely recognized, a price is also paid for them. First, by constraining the programmer with a type system, some apparently reasonable programs will be rejected even though their run-time behaviour would be acceptable. Second, the typing system may demand extensive programmer annotations, which can be time-consuming to write and tedious to read. Reducing the impact of these two drawbacks is the central objective of much of the work on types in programming languages.

A third motivation for types in programming languages is the one most recently understood: their role in supporting data abstraction and modu-

larity. Although these cornerstones of software engineering principle can be achieved to some extent without types, many programming languages employ a type system that enforces the hiding of data type representations and supports the specification of modules. For example, several languages support a separation between a package specification, which consists of a collection of type declarations, and the body of the package, which provides programs implementing the procedures, etc. that appear in the specification. These units are automatically analyzed to determine their type-correctness before code is generated by compilation.

A fourth motivation for the use of types, and the primary topic of this chapter, is their role as a conceptual tool for classifying programs in a way that permits a more abstract understanding of their meanings. For virtually any language, a semantics will classify objects according to their structure and use, thereby elevating them conceptually above the sequences of bits that they will be compiled into. This abstraction is the most fundamental use of type systems.

## 2 Types in programming

This section discusses a collection of programming examples intended to illustrate the motivations for various type structures. A handy pair of languages for making a comparison is Scheme, which can be viewed as based on the *untyped*  $\lambda$ -calculus, and ML, which can be viewed as based on the *typed*  $\lambda$ -calculus. Both languages have specifications that are at least as clear as one finds for most languages, so it is (usually) not difficult to tell what the meaning of a program in the language is actually specified to be. (An IEEE standard for Scheme was introduced in 1990 [IEE, 1991]; for ML the *Standard* version [Milner *et al.*, 1990] is used in the examples below.) While both were designed with semantic clarity as a key objective, the designs are also sensitive to efficiency issues, and both have good compilers that are widely used. Finally, the functional fragments of Scheme and ML employ a call-by-value evaluation strategy so the chance of confusing operational differences with differences in the type system philosophy are reduced.

### 2.1 Higher types

One of the first discoveries of researchers investigating the mathematical semantics of programming languages in the late 1960s was the usefulness of *higher-order* functions in describing the denotations of programs. The use of higher-order functions in *programming* was understood even earlier and incorporated in the constructs and programming methodology of Lisp. Programmers using the languages Scheme and ML employ higher-order functions as a tool for writing clear, succinct code by capturing good abstractions. On the other hand, the use of higher-order functions does

have its costs, and it is therefore worthwhile to discuss some of the ways in which such functions can be useful. Their usefulness as a tool in the semantics of programming languages is adequately argued by books and articles that employ them extensively in semantic descriptions. ([Tennent, 1992] provides a starting point.) Rather than review the subject from that perspective, let us instead consider briefly why they are useful in programming.

Consider a familiar mathematical operation: taking the derivative of a continuous real-valued function. The derivative of a function  $f$  is the function  $f'$  where

$$f'(x) = \frac{f(x + dx) - f(x)}{dx}$$

for an infinitesimal  $dx$ . For purposes of an estimate, it will simplify our discussion to bind  $dx$  to a small number (say 0.0001). Of course, it could passed as a parameter, but this is not necessary for the point below. A Scheme program for computing the derivative can be coded as follows:

```
(define (deriv f x)
  (/ (- (f (+ x dx)) (f x))
      dx))
```

Here the derivative is higher order, since it takes a function as a parameter, but it is coded as returning a numerical value on a given argument, rather than returning a new function as is the case for the mathematical derivative. This can lead to problems if the distinction is not properly observed. For example, the program `(deriv (deriv f 1))`, which might be mistakenly intended to compute the second derivative of  $f$  at 1, will yield a *run-time type error* complaining that `deriv` has the wrong number of arguments. Of course, the second derivative at 1 could be successfully calculated using an explicit abstraction,

```
(deriv (lambda (x) (deriv f x)) 1).
```

and the 'lambda' could be eliminated by making a local definition if this intrusion of lambda-abstraction is considered undesirable. However, these approaches generalize poorly to the case where what is wanted is the third derivative or the fourth derivative, and so on. To accommodate these cases, it would be possible to include a parameter  $n$  for the  $n$ th iteration of differentiation in the definition of `deriv`, but it is more elegant and understandable to quit fighting against the mathematical usage in the programming, and to start coding it instead. The derivative takes a function as an argument and produces a function as a value:

```
(define (deriv f)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x))
        dx)))
```



The second derivative at 1 is now properly coded as

```
((deriv (deriv f)) 1)
```

as in the mathematical notation  $f''(1)$ , where the primes denote an operation on a function. To calculate  $n$ th derivatives, it is possible to write another function that takes a function  $g$  and a number  $n$  as arguments and produces the function  $g^n = g \circ g \circ \dots \circ g$  ( $n$  copies) as its value. This is a powerful abstraction since there are many other ways this function might be used; the key idea here, the composition exponential, can be used modularly with the derivative rather than being mixed up in the code for the  $n$ th derivative function.

Where do types come into this? Lying at the heart of the distinction just discussed is the notion of ‘currying’, that is, the passage from a function of type  $r \times s \rightarrow t$  to one of type  $r \rightarrow (s \rightarrow t)$ . In the first case above, the derivative function was coded as a function taking as its arguments a real-valued function  $f$  and a real number  $x$ . When coded in ML, it looks like this:

```
fun deriv (f:real -> real, x:real):real
  = (f(x+dx) - f(x))/dx.
```

and the ML type-checker indicates its type as

```
((real -> real) * real) -> real
```

The way to read this is to think of it as the definition of a function `deriv` on a product type with the abstraction described using pattern matching. The second (curried) way to program this function is

```
fun deriv f = fn x:real => ((f(x+dx) - f(x))/dx):real
```

where the ML syntax for  $\lambda$  is `fn`. For this term, the type is

```
(real -> real) -> (real -> real)
```

Another example of the usefulness of higher-order functions comes from the powerful programming techniques one can obtain by combining them with references and assignments. An example drawn from [Abelson and Sussman, 1985] appears in Table 1. The procedure `make-account` takes a starting balance as an argument and produces an ‘account object’ as a value. The account object is itself a higher-order function, returned as `dispatch`, that has its own local state given by the contents of the ‘instance variable’ `balance`, which contains the current balance of the object. The arguments taken by the object include the ‘messages’ represented by atoms `withdraw` and `deposit` and the arguments of the ‘message sends’, which appear in the formal parameters `amount` in the ‘method definitions’ of `withdraw` and `deposit`. To create an account, the balance of the object must be initialized by applying `make-account` to the number that will be the starting value of its instance variable. For example, if `Dan` is defined as the value of `(make-account 50)` and `George` is defined as the value



**Table 1.** Using local variables and higher-order functions in scheme

---

```

(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (sequence (set! balance
                          (- balance amount))
                  balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request"
                        m))))
  dispatch)

```

---

of (make-account 100), then the account objects will correctly maintain their separate balance levels though a sequence of 'messages sends' describing the financial history of the objects. Finding a suitable system of types to classify programs written in 'object-oriented' style is a major area of research as this chapter is being written.

## 2.2 Recursive types

Consider the following pair of programs:

```

(define (cbvY f)
  ((lambda (x) (lambda (y) (f (x x) y)))
   (lambda (x) (lambda (y) (f (x x) y)))))

(define (badadd x) (+ x "astring")).

```

Both of them are compiled without comment by Scheme. They illustrate a trade-off in the type-checking of programs. The first program *cbvY* is the *call-by-value fixed-point combinator*. It is an interesting program that can be used to make recursive definitions without using explicit recursive equations. The second program *badadd* contains a 'semantic' error in the form of a bad addition. Many of the actual bugs encountered in programs are like this one—despite how silly it looks in this simple example. Both of these programs can be executed in Scheme although the second program will probably cause a run-time type error. These two examples could be rendered in ML as follows:

```

fun cbvY f =
  ((fn x => (fn y => f (x x) y))
   (fn x => (fn y => f (x x) y)))

fun badadd x = x + "astring".

```

but the ML type-checker will reject both of them as having type errors. In the second case the rejection will occur because there is an expression that purports to add something to a string—an operation that is not allowed. The first program is rejected because it has an application  $x\ x$  of a variable to itself and no type is inferred for this by the ML type-checker. From the viewpoint of a programmer, one might see the type-checking as a useful diagnostic for the second program and a hindrance for the first one.

A programming language is said to have *static* type-checking if the type-correctness of the program is established before the program is compiled. Scheme does not carry out such a check, but ML does. Of course, type-correctness is relative to the typing discipline, so this is a language design issue. In a language that allows most or all programs to be accepted as type-correct at compile time, it will be necessary to carry out various run-time type checks, and some errors that might have been caught by a type-checker may not be detected as quickly as one would desire. At an alternate extreme, a way to ensure that there are no run-time type errors is to reject all programs as having type errors at compile time. Of course, no programming language has a typing discipline as strict as that, but many languages are more restrictive than seems reasonable. The right balance in a language will discipline programming in order to provide useful diagnostic testing for errors while not ruling out programs that capture useful abstractions or efficient algorithms.

For example, languages that check the types of programs before compiling them can compensate for the problem with `cbvY` just mentioned by employing a more general type inference system than that of ML or by allowing the programmer to use explicit *recursive types*. The ML version of the call-by-value fixed-point combinator is given in Table 2. To understand this program, think of it as an annotation of the earlier program with coercions that make the types of subterms explicit. In fact, the type of the ML program is

$$((a \rightarrow b) \rightarrow (a \rightarrow b)) \rightarrow (a \rightarrow b)$$

rather than  $(c \rightarrow c) \rightarrow c$  (where  $a, b, c$  are type variables) as one might have hoped, but rather than carry out a detailed analysis of the program, let us just consider a couple of points about it. First of all, the definition of the procedure `cbvY` itself in the three lines of code between `in` and `end` is *not* recursive since `cbvY` appears only on the left-hand side of the defining equation. The recursion lies instead in the `datatype` declaration in the local bindings where a unary operator `fix` (written in postfix notation) is defined

**Table 2.** ML version of the call-by-value fixed-point combinator

---

```

local
  datatype 'a fix = FUN of 'a fix -> 'a
  fun NUF (FUN x) = x
in
  fun cbvY f =
    (fn x => (fn y => f((NUF x) x)y))
    (FUN(fn x => (fn y => f((NUF x) x)y)))
end

```

---

by a recursive equation. In that expression, the symbol 'a represents a *type variable*. A more mathematical way of writing the datatype declaration would be to indicate that  $fix(a) \cong fix(a) \rightarrow a$  where

$$FUN : (fix(a) \rightarrow a) \rightarrow fix(a)$$

defines the isomorphism. The inverse of the isomorphism is the function

$$NUF : fix(a) \rightarrow (fix(a) \rightarrow a)$$

defined in the third line of the program. If we remove the declaration of the type and the isomorphisms from this ML program, we obtain the definition of `cbvY` that was rejected as having type errors.

## 2.3 Parametric polymorphism

Although recursive types are a powerful tool for recovering the losses incurred by imposing a type discipline, there is another subtle concept to be found in the way certain abstractions can be formed in the untyped language. Here is a Scheme program that appends two lists of elements:

```

(define (append headlist taillist)
  (if (null? headlist)
      taillist
      (cons (car headlist)
            (append (cdr headlist) taillist))))

```

Scheme programmers never need to concern themselves about the *types* of the elements in the lists being appended, since this program will work equally well on any pair of arguments so long as they are *lists*. In some languages where programmers must declare their types, it is impossible to obtain this level of abstraction. Instead, it may be necessary to write a program that appends lists of integers and another program that appends lists of string arrays, and so on.

To avoid losing abstractions, languages with static type-checking deal with this problem by using *polymorphism*. The word 'polymorphism' means having many forms; in typed programming languages it ordinarily refers to the idea that a symbol may have many types. In one of its simplest forms, polymorphism arises from the use of a variable that specifies an indeterminate or parameterized type for an expression. This is called *parametric polymorphism*. A basic form of parametric polymorphism views this as a kind of macro expansion. For example, the Ada programming language has a construct known as a *generic* that serves this purpose. A procedure declared with a generic in Ada is explicitly instantiated with a type before it is used, but the abstraction can make it unnecessary to rewrite a piece of code that would work equally well for two different types. For example, the function that appends lists takes a pair of lists of elements of type  $t$  as an argument and returns a list of elements of type  $t$  as a result. Here the particular type  $t$  is unimportant, so it is replaced by a variable  $a$  and the type is indicated as  $\text{list}(a) \rightarrow \text{list}(a)$ . To see another example, consider the function that takes two reference cells and exchanges their contents. Obviously, this operation is independent of the types of elements in the reference cells; it is a 'polymorphic swapping' function. In ML it can be coded as follows:

```
fun swap (x,y) =  
  let val temp = ! x  
  in x := ! y ; y := temp  
  end
```

where the exclamation marks are the dereferencing operation:  $!x$  denotes the contents of the reference cell  $x$ . A novelty of the ML programming language is an inference algorithm that can *infer* a polymorphic type for programs without the need for programmer annotations. Specifically, using ML syntax, the type is inferred to be

```
swap : ('a ref * 'a ref) -> unit.
```

The function `swap` works with a side effect (change of memory); its output is unimportant so it is taken to be the unique value of type `unit`. The type of `swap` indicates that the references have the same type since the type variable `'a` is used for *both* arguments. This means that it is type-correct to swap the contents of two integer references or swap the contents of two string references, but a program that might swap the contents of an integer reference with that of a string reference will be rejected with a type error before being compiled.

Let us anticipate the precise definition of ML polymorphism with some discussion of what its limitations are in programming. Although type inference is an excellent tool for cutting the tedium of providing type annotations for programs, there is a great deal of abstraction that is lost in



the compromises of the ML polymorphic types. Consider, for example, the following Scheme program:

```
(define applyto
  (lambda (f) (cons (f 3) (f "hi"))))
```

It defines a procedure `applyto` which takes a function as an argument and forms a cons cell from the results of applying it to the number 3 and the string "hi". While it should not be difficult to think up many interesting things that can be applied to both 3 and "hi", let us keep things simple and consider

```
(applyto (lambda (x) x))
```

which evaluates to the cell (3 . "hi"). All this seems very simple and natural, but the ML type inference algorithm is unwilling to see this program as type-correct. In particular, the program

```
fn f => ( f(3), f("hi") )
```

will be rejected with an indication that the function `f` cannot take both 3 and "hi" as arguments since they have different types. This seems a bit dull-witted in light of the Scheme example, which evidently shows that there are perfectly good programs that *can* take both of these as arguments. ML has a construct that allows some level of such polymorphism. The program

```
let fun I x = x in (I(3), I("hi")) end
```

is type-correct but clearly fails to achieve the abstraction of the Scheme program since it only makes sense for a *given* value of `f` (in this case, `f` is `I`). To obtain a program as abstract as the one written in Scheme, it is necessary to introduce a more expressive type system than the one ML has. The Girard-Reynolds polymorphic  $\lambda$ -calculus, which is presented in a later section, has the desired expressiveness.

## 2.4 Subtypes

Another kind of programming language polymorphism that is being used in many modern languages is based on the notion of a *subtype*. This is a form of type polymorphism that arises from the classification of data according to collections of *attributes*. This perspective draws its inspiration from hierarchical systems of categories such as the taxonomy of the animal kingdom rather than from the variation of a parameter as in quantifiers of predicate logic.

To get some of the spirit of this kind of typing, let us begin with an informal example based on the kind of hierarchy that one might form in order to classify some of the individuals one finds at a university; let us call such individuals *academics*. Each academic has an associated *university* and *department* within the university. At the university there are *professors*, who teach courses, and *students*, who attend the courses taught by the



professors. Some of the students are *employees* of the university in a capacity as *teaching assistants (TAs)* while others are *research assistants (RAs)* supported by research grants. Each of these various classes of individuals has associated attributes. For instance, if we consider a typical semester, we can attribute to professors and teaching assistants the courses they are teaching—their *teaching load*. In this capacity as teachers, the professors and TAs are employees and therefore have a *salary* associated with them. RAs have a *project* attribute for the research project on which they are working.

To bring some order to this assortment of groups and attributes, it is helpful to organize a hierarchy of groups *classified by their defining attributes*. Let us begin to list each group and its attributes. First of all, we could use a type of *persons*, whose members, which include both academics and employees, have a *name* attribute. In addition to a name, each academic has a university and each employee has a *salary*, a *social security number* (for tax purposes), and an *employer*. In addition to attributes inherited from their roles as academics, each student has an *advisor* and each professor has a teaching load and a boolean *tenure* attribute. Professors are also employees, so they must possess the attributes of employees as well as those of academics. We can now classify our assortment by using common attributes to form the poset based on the relations

$$\begin{aligned} \text{Academic, Employee} &\leq \text{Person} \\ \text{Student, Professor} &\leq \text{Academic} \\ \text{RA, TA, Professor} &\leq \text{Employee} \\ \text{RA, TA} &\leq \text{Student} \end{aligned}$$

together with those relations obtained from an assumption that  $\leq$  is transitive and reflexive. Each point in the poset represents a *type* of individual based on attributes the individual must possess. If a type  $t$  is greater than a type  $s$  in the poset, this means that each kind of attribute that an individual of type of  $t$  possesses must also be had by each individual of type  $s$ . If  $s \leq t$ , we say that  $s$  is a *subtype* of  $t$ . The fact that a professor must have a social security number is something one can conclude by the fact that the type of professors is a subtype of that of employees and the fact that each employee has a social security number.

Each of the types in the example given above can be viewed as a kind of product where the components of a tuple having that type are its attributes. In programming languages these are generally called *records*, and the attributes are called the *fields* of the record. Records are usually written with curly brackets '{' and '}' rather than with parentheses as tuples are. Semantically they are very similar to tuples, but the field labels relieve the need to write the record fields in any particular order. A common record syntax is a sequence of pairs of the form  $l = M$ , where  $l$  is a *label* and  $M$

is the term associated with that label. The term  $M$  is generally called the *l-field* of the record. For example, the records

```
{Name = "Carl Gunter",
  University = "University of Pennsylvania"}
```

```
{University = "University of Pennsylvania",
  Name = "Carl Gunter"}
```

are considered equivalent, and the type of these records is given by the following equivalent pair of record type expressions:

```
{Name : String, University : String}
{University : String, Name : String}.
```

Now, we would like to mix records such as these with the dual notion of a *variant*. They are written with square (as opposed to curly) brackets '[' and ']'. For instance, a biological classification system might include a declaration such as

```
type ReproductiveSystem = [Male : MaleSystem,
                           Female : FemaleSystem]
```

defining a familiar partition of the collection of reproductive systems. In this expression, *Male* and *Female* are labels for the fields of the variant; the types of these fields must be *MaleSystem* and *FemaleSystem* respectively. The order in which the fields are written is insignificant. A classification system for vehicles might have a type

```
type Vehicle = [Air : AirVehicle,
                Land : LandVehicle,
                Water : WaterVehicle]
```

in which vehicles are classified according to their preferred milieu. A term of this type would come from one of the three possible components. For example,

```
[Air = SouthernCross]
```

is a term of type *Vehicle* if *SouthernCross* is a term of type *AirVehicle*. And

```
[Water = QueenMary]
```

is also a term of type *Vehicle* if *QueenMary* is a term of type *WaterVehicle*. To see a little more detail for these types, consider the declarations in Table 3. Here each of the defined types is a record type. To make the notation more succinct, a plus sign is written to indicate, for instance, that a *Machine* is a record having a field *Fuel* together with all of the fields had by a *Thing* (namely an *Age* field). Consider what a subtype of type *Vehicle* might be. In the case of records, a subtype has more fields than a supertype. In a variant, the dual holds. For instance,

**Table 3.** Declarations for a subtype hierarchy

---

```

type Thing = {Age : Int}
type Machine = Thing + {Fuel : String}
type MovingMachine = Machine + {MaxSpeed : Int}
type AirVehicle =
  MovingMachine +
  {MaxAltitude : Int, MaxPassengers : Int}
type LandVehicle =
  MovingMachine +
  {Surface : String, MaxPassengers : Int}
type WaterVehicle =
  MovingMachine +
  {Tonnage : Int, MaxPassengers : Int}

```

---

```

type WheeledVehicle = [Air : WheeledAirVehicle,
                       Land : WheeledLandVehicle]

```

is a subtype of `Vehicle` where

```

type WheeledLandVehicle = LandVehicle +
  {WheelsNumber : Int}
type WheeledAirVehicle = AirVehicle +
  {WheelsNumber : Int}.

```

Intuitively, a wheeled vehicle is either an air vehicle with wheels or a land vehicle with wheels. If we forget about the wheels, then a wheeled vehicle can be viewed simply as a vehicle. This example also illustrates that it is not just the fact that there are fewer fields that matters for variants, but that the types of the fields that exist are subtypes of the corresponding fields from the supertype. Looking at this from the point of view of a term of type `WheeledVehicle`, note that

```

value MyCar = [Land = {Age = 3,
                      Fuel = "Gasoline",
                      MaxSpeed = 100,
                      Surface = "Roadway",
                      MaxPassengers = 5,
                      WheelsNumber = 4}]

```

has the type `Vehicle` if the last field, which indicates the number of wheels, is omitted.

This provides some intuition about the subtyping relation between records and between variants, but there is still one more type constructor to which we would like to generalize the idea: the function space operator. Suppose, for instance, that we need a function

Using : String -> Machine

which, given a kind of fuel (described by a string), returns an example of a Machine that uses that fuel. In any context where such a function is needed, we could just as easily use a function

WaterVehicleUsing : String -> WaterVehicle,

which, given a kind of fuel (described by a string), returns an example of a WaterVehicle that uses that fuel. This suffices because a WaterVehicle is a kind of machine.

Suppose now that we need a function having the type

HowSoon : {Start : Place, Finish : Place,  
          Mode : AirVehicle} -> Int

where the type Place is a record consisting of a latitude and a longitude and the function calculates a lower bound on how soon the given mode of transport could make it from Start to Finish. Suppose we have on hand a function

MovingMachineHowSoon : {Start : Place, Finish : Place,  
                          Mode : MovingMachine} -> Int

which calculates a value from its arguments in the naive way, using the distance between the two places and the maximum speed of a MovingMachine as an argument. This can be used to serve the purpose of HowSoon since an AirVehicle is a special kind of MovingMachine. The method used to calculate HowSoon on an instance of the latter type also applies to an instance of the former.

These examples suggest that we should take String -> Machine to be a subtype of String -> WaterVehicle and take

{Start : Place, Finish : Place, Mode : MovingMachine}  
-> Int

to be a subtype of

{Start : Place, Finish : Place, Mode : AirVehicle}  
-> Int.

In the general case we will want to generalize this by taking  $s \rightarrow t$  to be a subtype of  $s' \rightarrow t'$  just in case  $t$  is a subtype of  $t'$  and  $s'$  is a subtype of  $s$ . Note the change in the ordering with respect to the first arguments: if  $s \rightarrow t \leq s' \rightarrow t'$ , then  $s' \leq s$  rather than  $s \leq s'$ .

### 3 Simple types as sets

The simply-typed  $\lambda$ -calculus is the most basic of the typed calculi with higher-order functions. It is described as a collection of terms and types together with independent systems of typing judgements and equational judgements. The types  $t$  and terms  $M$  are given by the following grammar:



$$\begin{array}{lll}
x & \in & \text{Variable} \\
t & ::= & \mathbf{o} \mid t \rightarrow t \\
M & ::= & x \mid \lambda x : t. M \mid MM
\end{array}$$

where *Variable* is a (possibly infinite) collection of primitive syntactic objects called *variables*. In the discussions below, letters from the end of the alphabet such as  $x, y, z$  and such letters with subscripts and superscripts as in  $x', x_1, x_2$  range over variables, but it is also handy to use letters such as  $f, g$  for variables in some cases. Types are generally written using letters  $r, s$ , and  $t$ . Terms are generally written with letters  $L, M, N$ . Such letters annotated with superscripts and subscripts may also be used when convenient. The type  $\mathbf{o}$  is called the *ground type*, and types  $s \rightarrow t$  are called *higher types*. Terms of the form  $\lambda x : t. M$  are called *abstractions*, and those of the form  $MM$  are called *applications*.

Parentheses are used to indicate how an expression is parsed modulo some standard parsing conventions. For types, the *association of the operator  $\rightarrow$  is to the right*: for instance,  $\mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}$  parses as  $\mathbf{o} \rightarrow (\mathbf{o} \rightarrow \mathbf{o})$ . Dually, *application operations associate to the left*: an application  $LMN$  should be parsed as  $(LM)N$ . So, the expression  $xyz$  unambiguously parses as  $(xy)z$ . If we wish to write the expression that applies  $x$  to the result of applying  $y$  to  $z$ , it is rendered as  $x(yz)$ . Moreover *application binds more tightly than abstraction*: for instance, an expression  $\lambda x : t. MN$  should be parsed as  $\lambda x : t. (MN)$ . Hence, the expression  $\lambda x : s. \lambda y : t. xyz$  unambiguously parses as  $\lambda x : s. \lambda y : t. ((xy)z)$ . Superfluous parentheses can be sprinkled into an expression at will to emphasize grouping. There is no distinction between  $M$  and  $(M)$ , and it is common to surround the operand of an application  $M(N)$  with parentheses to mimic the mathematical notation  $f(x)$  for a function applied to an argument.

Terms are treated as equivalent up to the renaming of bound variables ( $\alpha$ -equivalence). To avoid tedious repetitions of assumptions about the names of bound variables, it is helpful to use the

**Bound variable naming convention** When a term representing an  $\alpha$ -equivalence class is chosen, the name of the bound variable of the representative is taken to be distinct from the names of free variables in other terms being discussed.

Syntactic identity between terms is denoted by the relation  $\equiv$ . Given terms  $M$  and  $N$  and a variable  $x$ , the expression  $[M/x]N$  is the term obtained by substituting  $M$  for  $x$  in  $N$ . This must be done modulo renaming bound variables in  $N$  to avoid capturing free variables of  $M$ .

### 3.1 Types and equations

To describe the typing system for the simply-typed  $\lambda$ -calculus, some notation for associating types with free variables is required. A *type assignment* is a list  $H \equiv x_1 : t_1, \dots, x_n : t_n$  of pairs of variables and types such that the



variables  $x_i$  are distinct. The empty type assignment  $\emptyset$  is the degenerate case in which there are no pairs. Write  $x : t \in H$  if  $x$  is  $x_i$  and  $t$  is  $t_i$  for some  $i$ . In this case it is said that  $x$  *occurs* (or *appears*) in  $H$ , and this may be abbreviated by writing  $x \in H$ . If  $x : t \in H$ , then define  $H(x)$  to be the type  $t$ .

A *typing judgement* is a triple consisting of a type assignment  $H$ , a term  $M$ , and a type  $t$  such that all of the free variables of  $M$  appear in  $H$ . This relation between  $H$ ,  $M$ , and  $t$  is written in the form  $H \vdash M : t$  and read ‘in the assignment  $H$ , the term  $M$  has type  $t$ ’. It is defined to be the least relation satisfying the axiom and two rules in Table 4. A demonstration

**Table 4.** Typing rules for the simply-typed  $\lambda$ -calculus

[Proj]	$H, x : t, H' \vdash x : t$
[Abs]	$\frac{H, x : s \vdash M : t}{H \vdash \lambda x : s. M : s \rightarrow t}$
[Appl]	$\frac{H \vdash M : s \rightarrow t \quad H \vdash N : s}{H \vdash M(N) : t}$

of  $H \vdash M : t$  from these rules is called a *typing derivation*. We have the following basic fact about this system:

**Lemma 3.1.1.** *If  $H \vdash M : t$  and  $x$  does not appear in  $H$ , then  $x$  is not free in  $M$ .*

In general, we will only be interested in terms  $M$  and type assignments  $H$  such that  $H \vdash M : t$  for some type  $t$ . A term  $M$  is said to be *untypeable* if there is no type assignment  $H$  and type  $t$  such that  $H \vdash M : t$ . For example,  $\lambda x : \mathbf{o}. x(x)$  fails to have a type in any type assignment. If a term has a type in a given assignment, that type is unique in the following sense:

**Lemma 3.1.2.** *If  $H \vdash M : s$  and  $H \vdash M : t$ , then  $s \equiv t$ .*

Type tags are placed on bound variables in abstractions just to make Lemma 3.1.2 true. If we try to simplify our notation by allowing terms of the form  $\lambda x. M$  and a typing rule of the form

$$[\text{Abs}]^- \quad \frac{H, x : s \vdash M : t}{H \vdash \lambda x. M : s \rightarrow t}$$

then Lemma 3.1.2 would *fail*. For example, we would then have

$$\vdash \lambda x. x : \mathbf{o} \rightarrow \mathbf{o}$$

as well as

$$\vdash \lambda x. x : (\mathbf{o} \rightarrow \mathbf{o}) \rightarrow (\mathbf{o} \rightarrow \mathbf{o}).$$

Some further important properties of the type system are the following:

**Lemma 3.1.3.** *If  $H, x : r, y : s, H' \vdash M : t$ , then  $H, y : s, x : r, H' \vdash M : t$ .*

**Lemma 3.1.4.** *If  $H, x : s, H' \vdash M : t$  and  $H, H' \vdash N : s$ , then  $H, H' \vdash [N/x]M : t$ .*

An equation in the simply-typed lambda-calculus is a four-tuple

$$(H, M, N, t),$$

where  $H$  is a type assignment,  $M, N$  are  $\lambda$ -terms, and  $t$  is a type. To make a tuple like this more readable, it is helpful to replace the commas separating the components of the tuple by more suggestive symbols and to write

$$(H \triangleright M = N : t).$$

The triangular marker is intended to indicate where the interesting part of the tuple begins. The heart of the tuple is the pair of terms on either side of the equation symbol;  $H$  and  $t$  provide typing information about these terms. An *equational theory*  $T$  is a set of equations  $(H \triangleright M = N : t)$  such that  $H \vdash M : t$  and  $H \vdash N : t$ . An equation  $(H \triangleright M = N : t)$  should be viewed only as a formal symbol. For the judgement that an equation is *provable*, we define the relation  $\vdash$  between theories  $T$  and equations  $(H \triangleright M = N : t)$  to be the least relation satisfying the rules in Table 5.

The assertion  $T \vdash (H \triangleright M = N : t)$  is called an *equational judgement*. Of course, the turnstile symbol  $\vdash$  is also used for typing judgements, but this overloading is never a problem because of the different appearance of the two forms of judgement. The two are related by the following fact:

**Lemma 3.1.5.** *If  $T$  is a theory and  $T \vdash (H \triangleright M = N : t)$ , then  $H \vdash M : t$  and  $H \vdash N : t$ .*

Another basic property is the following:

**Lemma 3.1.6.** *Suppose  $H \vdash M : t$  and  $H \vdash N : t$ . Let  $H'$  be a type assignment such that  $H'(x) = H(x)$  for each  $x \in \text{Fv}(M) \cup \text{Fv}(N)$ . If  $T \vdash (H' \triangleright M = N : t)$ , then also  $T \vdash (H \triangleright M = N : t)$ .*

Further discussion of the typed  $\lambda$ -calculus, including many of its interesting syntactic properties, can be found in [Hindley and Seldin, 1986] or in [Barendregt, 1992].

**Table 5.** Equational rules for the simply-typed  $\lambda$ -calculus

---

{Axiom}	$\frac{(H \triangleright M = N : t) \in T}{T \vdash (H \triangleright M = N : t)}$
{Add}	$\frac{T \vdash (H \triangleright M = N : t) \quad x \notin H}{T \vdash (H, x : s \triangleright M = N : t)}$
{Drop}	$\frac{T \vdash (H, x : s \triangleright M = N : t) \quad x \notin \text{Fv}(M) \cup \text{Fv}(N)}{T \vdash (H \triangleright M = N : t)}$
{Permute}	$\frac{T \vdash (H, x : r, y : s, H' \triangleright M = N : t)}{T \vdash (H, y : s, x : r, H' \triangleright M = N : t)}$
{Ref}	$\frac{H \vdash M : t}{T \vdash (H \triangleright M = M : t)}$
{Sym}	$\frac{T \vdash (H \triangleright M = N : t)}{T \vdash (H \triangleright N = M : t)}$
{Trans}	$\frac{T \vdash (H \triangleright L = M : t) \quad T \vdash (H \triangleright M = N : t)}{T \vdash (H \triangleright L = N : t)}$
{Cong}	$\frac{T \vdash (H \triangleright M = M' : s \rightarrow t) \quad T \vdash (H \triangleright N = N' : s)}{T \vdash (H \triangleright M(N) = M'(N') : t)}$
{ $\xi$ }	$\frac{T \vdash (H, x : s \triangleright M = N : t)}{T \vdash (H \triangleright \lambda x : s. M = \lambda x : s. N : s \rightarrow t)}$
{ $\beta$ }	$\frac{H, x : s \vdash M : t \quad H \vdash N : s}{T \vdash (H \triangleright (\lambda x : s. M)(N) = [N/x]M : t)}$
{ $\eta$ }	$\frac{H \vdash M : s \rightarrow t \quad x \notin \text{Fv}(M)}{T \vdash (H \triangleright \lambda x : s. M(x) = M : s \rightarrow t)}$

---

### 3.2 Sets as a model

The ‘standard’ model of the simply-typed  $\lambda$ -calculus interprets types as sets where higher types are the sets of functions between sets. The semantics is relative to the choice of a set  $X$ , which serves as the interpretation for

the base type. The meaning  $\llbracket t \rrbracket$  of a type  $t$  is a set defined inductively as follows:

- $\llbracket \mathbf{o} \rrbracket = X$
- $\llbracket s \rightarrow t \rrbracket = \{f \mid f \text{ is a function from } \llbracket s \rrbracket \text{ to } \llbracket t \rrbracket\}$ .

So, for example,  $\llbracket \mathbf{o} \rightarrow (\mathbf{o} \rightarrow \mathbf{o}) \rrbracket$  is the set of functions  $f$  such that, for each  $x \in X$ ,  $f(x)$  is a function from  $X$  into  $X$ . On the other hand,  $\llbracket (\mathbf{o} \rightarrow \mathbf{o}) \rightarrow \mathbf{o} \rrbracket$  is the set of functions  $F$  such that, for each function  $f$  from  $X$  to  $X$ ,  $F(f)$  is an element of  $X$ .

Describing the meanings of terms is more difficult than describing the meanings of types, and we require some further vocabulary and notation. While a type assignment associates *types* with variables, an *environment* associates *values* to variables. Environments are classified by type assignments: if  $H$  is a type assignment, then an  $H$ -environment is a function  $\rho$  on variables that maps each  $x \in H$  to a value  $\rho(x) \in \llbracket H(x) \rrbracket$ . If  $\rho$  is an  $H$ -environment,  $x : t \in H$ , and  $d \in \llbracket t \rrbracket$ , then we define

$$\rho[x \mapsto d](y) = \begin{cases} d & \text{if } y \equiv x \\ \rho(y) & \text{otherwise.} \end{cases}$$

This is the ‘update’ operation. One can read  $\rho[x \mapsto d]$  as ‘the environment  $\rho$  with the value of  $x$  updated to  $d$ ’. The notation is similar to that used for syntactic substitution, but note that this operation on environments is written as a postfix. So another way to read  $\rho[x \mapsto d]$  is ‘the environment  $\rho$  with  $d$  for  $x$ .’ Note that if  $x \notin H$  for an assignment  $H$ , then  $\rho[x \mapsto d]$  is an  $H, x : t$  environment if  $d \in \llbracket t \rrbracket$ . Now, the meaning of a term  $M$  is described relative to a type assignment  $H$  and a type  $t$  such that  $H \vdash M : t$ . We use the notation  $\llbracket H \triangleright M : t \rrbracket$  for the meaning of term  $M$  relative to  $H, t$ . Here, as in the case of equations earlier, the triangle is intended as a kind of marker or separator between the type assignment  $H$  and the term  $M$ . We might have written  $\llbracket H \vdash M : t \rrbracket$  for the meaning, but this confuses the use of  $\vdash$  as a relation for typing judgements with its syntactic use as a punctuation in the expression within the semantic brackets. Nevertheless, it is important to remember that  $\llbracket H \triangleright M : t \rrbracket$  only makes sense if  $H \vdash M : t$ .

The meaning  $\llbracket H \triangleright M : t \rrbracket$  is a function from  $H$ -environments to  $\llbracket t \rrbracket$ . The semantics is defined by induction on the typing derivation of  $H \vdash M : t$ ,

- Projection:  $\llbracket H \triangleright x : t \rrbracket \rho = \rho(x)$ .
- Abstraction:  $\llbracket H \triangleright \lambda x : u. M' : u \rightarrow v \rrbracket \rho$  is the function from  $\llbracket u \rrbracket$  to  $\llbracket v \rrbracket$  given by  $d \mapsto \llbracket H, x : u \triangleright M' : v \rrbracket (\rho[x \mapsto d])$ , that is, the function  $f$  defined by

$$f(d) = \llbracket H, x : u \triangleright M' : v \rrbracket (\rho[x \mapsto d]).$$

- Application:  $\llbracket H \triangleright L(N) : t \rrbracket \rho$  is the value obtained by applying the function  $\llbracket H \triangleright L : s \rightarrow t \rrbracket \rho$  to argument  $\llbracket H \triangleright N : s \rrbracket \rho$  where  $s$  is the unique type such that  $H \vdash L : s \rightarrow t$  and  $H \vdash N : s$ .

It will save us quite a bit of ink to drop the parentheses that appear as part of expressions such as  $\llbracket H, x : u \triangleright M' : v \rrbracket (\rho[x \mapsto d])$  and simply write  $\llbracket H, x : u \triangleright M' : v \rrbracket \rho[x \mapsto d]$ . Doing so appears to violate the convention of associating applications to the left, but there is little chance of confusion in the case of expressions such as these. Hence, we will adopt the convention that the postfix update operator binds more tightly than general application.

It can be shown that this assignment of meanings respects our equational rules. This is the *soundness* property of the semantic interpretation:

**Theorem 3.2.1 (Soundness).** *If  $\vdash (H \triangleright M = N : t)$ , then  $\llbracket H \triangleright M : t \rrbracket = \llbracket H \triangleright N : t \rrbracket$ .*

This is proved by induction on the height of a derivation tree for an equational judgement by examining each case for the last rule employed. For example, the soundness of the  $\eta$ -rule depends on the following fact:

**Lemma 3.2.2.** *Suppose  $M$  is a term and  $H \vdash M : t$ . If  $x \notin H$  and  $d \in \llbracket s \rrbracket$ , then  $\llbracket H, x : s \triangleright M : t \rrbracket \rho[x \mapsto d] = \llbracket H \triangleright M : t \rrbracket \rho$ .*

The lemma essentially asserts that the meaning of a term  $M$  in a type environment  $H$  depends only on the values  $H$  assigns to free variables of  $M$ . We may therefore calculate

$$\begin{aligned}
 & \llbracket H \triangleright \lambda x : s. M(x) : s \rightarrow t \rrbracket \rho \\
 &= (d \mapsto \llbracket H, x : s \triangleright M(x) : t \rrbracket \rho[x \mapsto d]) \\
 &= (d \mapsto (\llbracket H, x : s \triangleright M : s \rightarrow t \rrbracket \rho[x \mapsto d])(d)) \\
 &= (d \mapsto (\llbracket H \triangleright M : s \rightarrow t \rrbracket \rho)(d)) \\
 &= \llbracket H \triangleright M : s \rightarrow t \rrbracket \rho
 \end{aligned}$$

where the third equality follows from Lemma 3.2.2.

As an application of the soundness of our interpretation, consider the following:

**Theorem 3.2.3.** *The simply-typed  $\lambda$ -calculus is non-trivial. That is, for any type  $t$  and pair of distinct variables  $x$  and  $y$ , it is not the case that  $\vdash (x : t, y : t \triangleright x = y : t)$ .*

**Proof.** Suppose, on the contrary, that  $\vdash (x : t, y : t \triangleright x = y : t)$ . Let  $X$  be any set with more than one element and consider the model of the simply-typed  $\lambda$ -calculus generated by  $X$ . It is not hard to see that  $\llbracket t \rrbracket$  has at least two distinct elements  $p$  and  $q$ . Now, let  $\rho$  be an  $x : t, y : t$  environment such that  $\rho(x) = p$  and  $\rho(y) = q$ . Then  $\llbracket x : t, y : t \triangleright x = y : t \rrbracket \rho$



$= \rho(x) = p \neq q = \rho(y) = \llbracket x : t, y : t \triangleright y : t \rrbracket \rho$ . But this contradicts the soundness of our interpretation. ■

It is instructive, as an exercise on the purpose of providing a semantic interpretation for a calculus, to try proving Theorem 3.2.3 directly from first principles and the rules for the  $\lambda$ -calculus using syntactic means. The soundness result provides us with a simple way of demonstrating properties of the rules of our calculus or, dually, a syntax for proving properties of our model (sets and functions).

### 3.3 Type frames

Although we have given a way to associate a ‘meaning’  $\llbracket H \triangleright M : t \rrbracket$  to a triple  $H, M, t$  such that  $H \vdash M : t$  and demonstrated that our assignment of meaning preserves the required equations from Table 5, we did not actually provide a rigorous description of the ground rules for saying when such an assignment really is a *model* of the simply-typed  $\lambda$ -calculus. In fact, there is more than one way to do this, depending on what one considers important about the model. The choice of definition may be a matter of style or convenience, but different choices may also reflect significant distinctions. The form of model described in this section is generally referred to as an ‘extensional environment model’. The discussion follows the treatment of [Friedman, 1975] and the primary objective is to discuss the two completeness theorems that he proves there (given as Theorems 3.3.8 and 3.4.5 below).

For the sake of convenience, the definition is broken into two parts. Models are called frames; these are defined in terms of a more general structure called a pre-frame.

**Definition 3.3.1.** A *pre-frame* is a pair of functions  $\mathcal{A}[\cdot]$  and  $A$  on types and pairs of types respectively such that

- $\mathcal{A}[t]$  is a non-empty set, which we view as the interpretation of type  $t$ , and
- $A^{s,t} : \mathcal{A}[s \rightarrow t] \times \mathcal{A}[s] \rightarrow \mathcal{A}[t]$  is a function that we view as the interpretation of the application of an element of  $\mathcal{A}[s \rightarrow t]$  to an element of  $\mathcal{A}[s]$ ,

and such that the *extensionality property* holds: that is, whenever  $f, g \in \mathcal{A}[s \rightarrow t]$  and  $A^{s,t}(f, x) = A^{s,t}(g, x)$  for every  $x \in \mathcal{A}[s]$ , then  $f = g$ .

To make the notation less cumbersome, we write  $(\mathcal{A}, A)$  for a pre-frame and use  $\mathcal{A}$  to represent the pair. Pre-frames are very easy to find. For example, we might take  $\mathcal{A}[s]$  to be the set of natural numbers for every  $s$  and define  $A^{s,t}(f, x)$  to be the product of  $f$  and  $x$ . Since  $f * 1 = g * 1$  implies  $f = g$ , the extensionality property is clearly satisfied. Nevertheless, this multiplication pre-frame does not provide any evident interpretation for  $\lambda$ -terms (indeed, the reader may wish to try the exercise of proving that

there is none satisfying the equational rules).

A frame is a pre-frame together with a sensible interpretation for  $\lambda$ -terms.

**Definition 3.3.2.** A *type frame* (or *frame*) is a pre-frame  $(\mathcal{A}^{\text{type}}, A)$  together with a function  $\mathcal{A}^{\text{term}}$  defined on triples  $H \triangleright M : t$  such that  $H \vdash M : t$ . An *H-environment* is a function  $\rho$  from variables to meanings such that  $\rho(x) \in \mathcal{A}^{\text{type}}[H(x)]$  whenever  $x \in H$ .  $\mathcal{A}^{\text{term}}[H \triangleright M : t]$  is a function from  $H$ -environments into  $\mathcal{A}^{\text{type}}[t]$ . The function  $\mathcal{A}^{\text{term}}[\cdot]$  is required to satisfy the following equations:

1.  $\mathcal{A}^{\text{term}}[H \triangleright x : t]\rho = \rho(x)$
2.  $\mathcal{A}^{\text{term}}[H \triangleright M(N) : t]\rho = A^{s,t}(\mathcal{A}^{\text{term}}[H \triangleright M : s \rightarrow t]\rho, \mathcal{A}^{\text{term}}[H \triangleright N : s]\rho)$
3.  $A^{s,t}(\mathcal{A}^{\text{term}}[H \triangleright \lambda x : s. M : s \rightarrow t]\rho, d) = \mathcal{A}^{\text{term}}[H, x : s \triangleright M : t]\rho[x \mapsto d]$ .

If a pre-frame has an extension to a frame, then the extension is unique.

**Lemma 3.3.3.** Let  $(\mathcal{A}^{\text{type}}, A)$  be a pre-frame over which  $\mathcal{A}^{\text{term}}[\cdot]$  and  $\bar{\mathcal{A}}^{\text{term}}[\cdot]$  define frames. Then  $\mathcal{A}^{\text{term}}[H \triangleright M : t] = \bar{\mathcal{A}}^{\text{term}}[H \triangleright M : t]$  whenever  $H \vdash M : t$ .

In general, it is therefore convenient to use the same notation  $\mathcal{A}$  for both  $\mathcal{A}^{\text{type}}[\cdot]$  and  $\mathcal{A}^{\text{term}}[\cdot]$ . The lemma says that the former together with an application operation  $A$  determines the latter, so it simplifies matters to write a pair  $(\mathcal{A}, A)$  for a frame.

A frame  $\mathcal{A}$  should be viewed as a model of the  $\lambda$ -calculus; we write

$$\mathcal{A} \models (H \triangleright M = N : t)$$

if, and only if,  $\mathcal{A}[H \triangleright M : t]\rho = \mathcal{A}[H \triangleright N : t]\rho$  for each  $H$ -environment  $\rho$ . Whenever it will not cause confusion, it helps to drop the typing information and write  $\mathcal{A} \models M = N$ . If  $T$  is a set of equations, then

$$\mathcal{A} \models T$$

if, and only if,  $\mathcal{A} \models (H \triangleright M = N : t)$  for each equation  $(H \triangleright M = N : t)$  in  $T$ . Define  $T \models M = N$  if  $\mathcal{A} \models M = N$  whenever  $\mathcal{A} \models T$ .

The ‘standard’ frame uses sets and functions: given a set  $X$ , the *full frame over  $X$*  is  $\mathcal{F}_X = (\mathcal{F}_X[\cdot], F_X)$  where

- $\mathcal{F}_X[\mathbf{o}] = X$  and  $\mathcal{F}_X[s \rightarrow t]$  is the set of functions from  $\mathcal{F}_X[s]$  to  $\mathcal{F}_X[t]$
- $F_X^{s,t}(f, x) = f(x)$ , that is,  $F_X^{s,t}$  is ordinary function application,
- on terms,  $\mathcal{F}_X[\cdot]$  is the function  $[\cdot]$  defined in the previous section.

It is easy to see that our definition of the semantic function  $[\cdot]$  corresponds exactly to the three conditions in the definition of a frame. Moreover,

these were essentially the properties that made our proof of the soundness property for the interpretation possible. To be precise:

**Theorem 3.3.4 (Soundness for frames).** *For any theory  $T$  and frame  $\mathcal{A}$ , if  $\mathcal{A} \models T$  and  $T \vdash (H \triangleright M = N : t)$ , then  $\mathcal{A} \models (H \triangleright M = N : t)$ .*

When  $T$  is empty, we have the following:

**Corollary 3.3.5.** *For any frame  $\mathcal{A}$ , if  $\vdash M = N$ , then  $\mathcal{A} \models M = N$*

Another important class of examples of frames can be formed from equivalence classes of well-typed terms of the simply-typed calculus. To define these frames we need some more notation for type assignments. An *extended* type assignment  $\mathcal{H} = x_1 : t_1, x_2 : t_2, \dots$  is an infinite list of pairs such that every finite prefix  $H \subseteq \mathcal{H}$  is a type assignment and every type appears infinitely often (that is, for each type  $t$ , there are infinitely many variables  $x$  such that  $x : t$  appears in  $\mathcal{H}$ ). Note that if  $H \vdash M : t$  and  $H' \vdash M : s$  where  $H, H' \subseteq \mathcal{H}$ , then  $s \equiv t$ . Now, fix an extended type assignment  $\mathcal{H}$ . Let us say that a theory  $T$  is an  $\mathcal{H}$ -theory if  $H \subseteq \mathcal{H}$  for each  $(H \triangleright M = N : t) \in T$ . Let  $T$  be an  $\mathcal{H}$ -theory. If  $H \vdash M : t$  for some  $H \subseteq \mathcal{H}$ , define

$$[M]_T = \{M' \mid T \vdash (H' \triangleright M = M' : t) \text{ for some } H' \subseteq \mathcal{H}\}.$$

This defines an equivalence relation on such terms  $M$  (the proof is left as an exercise). When  $T$  is the empty set, we drop the subscript  $T$ . For each type  $t$ , define

$$\mathcal{T}_T[t] = \{[M]_T \mid H \vdash M : t \text{ for some } H \subseteq \mathcal{H}\}.$$

For each pair of types  $s, t$ , define  $\text{TermAppl}_T^{s,t} : \mathcal{T}_T[s \rightarrow t] \times \mathcal{T}_T[s] \rightarrow \mathcal{T}_T[t]$  by

$$\text{TermAppl}_T^{s,t}([M]_T, [N]_T) = [M(N)]_T.$$

This is well-defined because of the congruence rule for application. It can be shown that

**Lemma 3.3.6.** *The pair  $(\mathcal{T}_T, \text{TermAppl}_T)$  is a pre-frame.*

Indeed, this pre-frame is a frame, which is called the *term model* over  $T$ . To see this we need a notation for *simultaneous* substitutions. We write  $\sigma = [M_1, \dots, M_n/x_1, \dots, x_n]$  for the function that maps the variable  $x_i$  to the term  $M_i$  for each  $i$  and acts as the identity on other variables. It is assumed that  $x_1, \dots, x_n$  are distinct. The *support* of the substitution is the set of variables on which the substitution is not the identity; of course, the support of  $[M_1, \dots, M_n/x_1, \dots, x_n]$  is a subset of  $\{x_1, \dots, x_n\}$ . The substitution  $\sigma = [M_1, \dots, M_n/x_1, \dots, x_n]$  can be extended to substitution on terms by inductively defining

- $\sigma(M(N)) \equiv (\sigma(M))(\sigma(N))$

- $\sigma(\lambda x : t. M) \equiv \lambda x : t. \sigma(M)$  where  $x$  is not in the support of  $\sigma$  or in  $\text{Fv}(\sigma(y))$  for any  $y$  in the support of  $\sigma$ .

This generalizes our earlier notation  $[M/x]$  which may now be viewed as a substitution with support  $\{x\}$ . When  $x$  is not in the support of  $\sigma$ , we write  $\sigma[x \mapsto M]$  or  $\sigma[M/x]$  for  $[M_1, \dots, M_n, M/x_1, \dots, x_n, x]$ .

Let  $\rho$  be an  $H$ -environment for the term pre-frame: that is,  $\rho(x) \in \mathcal{T}_T[H(x)]$  whenever  $x \in H$ . Let us say that a substitution  $\sigma$  *represents*  $\rho$  over  $H$  if, for each  $x$  in  $H$ , the term  $\sigma(x)$  is a representative of the term model equivalence class  $\rho(x)$ .

**Lemma 3.3.7.** *Let  $\mathcal{T}_T[H \triangleright M : t]\rho = [\sigma(M)]_T$  where  $\sigma$  is a substitution representing  $\rho$  over  $H$ . Then  $(\mathcal{T}_T, \text{TermAppl}_T)$  is a type frame.*

Type frames form a complete class of models for theories of the simply-typed calculus:

**Theorem 3.3.8 (Completeness for frames).**  *$T \vdash M = N$  if, and only if,  $T \models M = N$ .*

This follows immediately from Lemma 3.3.7 and the following:

**Theorem 3.3.9.** *Suppose  $H \subseteq \mathcal{H}$  and  $T$  is an  $\mathcal{H}$ -theory, then  $T \vdash (H \triangleright M = N : t)$  if, and only if,  $\mathcal{T}_T \models (H \triangleright M = N : t)$ .*

**Proof.** Necessity follows immediately from the soundness Theorem 3.3.4 for frames and the fact that the term model is a frame. To prove sufficiency, choose  $\rho$  to be the ‘identity’ environment  $\rho : x \mapsto [x]_T$ . The identity substitution  $\sigma : x \mapsto x$  represents this over  $H$ . Now,  $[M]_T = [\sigma(M)]_T = \mathcal{T}_T[H \triangleright M : t]\rho = \mathcal{T}_T[H \triangleright N : t]\rho = [\sigma(N)]_T = [N]_T$  so  $T \vdash (H' \triangleright M = N : t)$  for some  $H' \subseteq \mathcal{H}$ . Hence, by Lemma 3.1.6,  $T \vdash (H \triangleright M = N : t)$  as well. ■

A particularly important example of a frame in the class of term models is the one induced by the empty theory:  $\mathcal{T}_\emptyset$ . For this particular term model it is convenient to drop the subscript  $\emptyset$ . As an instance of Theorem 3.3.9, we have the following:

**Corollary 3.3.10.**  *$\vdash M = M'$  if, and only if,  $\mathcal{T} \models M = M'$ .*

### 3.4 Completeness for sets

Given a collection of mathematical structures, it is usually fruitful to find and study collections of structure-preserving transformations or mappings between them. Homomorphisms of algebras are one such example, and continuous maps on the real numbers another example. What kinds of mappings between type frames should we take to be ‘structure-preserving’? The definition we seek for the goal of this section is obtained by following the spirit of homomorphisms between algebras but permitting *partial* structure-preserving mappings and requiring such maps to be surjective.



This will provide the concept needed to prove that the full type frame is complete.

**Definition 3.4.1.** Let  $\mathcal{A}$  and  $\mathcal{B}$  be frames. A *partial homomorphism*  $\Phi : \mathcal{A} \rightarrow \mathcal{B}$  is a family of surjective partial functions  $\Phi^s$  from  $\mathcal{A}[s]$  into  $\mathcal{B}[s]$  such that, for each  $s, t$ , and  $f \in \mathcal{A}[s \rightarrow t]$  either

1. there is some  $g \in \mathcal{B}[s \rightarrow t]$  such that

$$\Phi^t(A^{s,t}(f, x)) = B^{s,t}(g, \Phi^s(x)) \quad (3.1)$$

for all  $x$  in the domain of definition of  $\Phi^s$  and  $\Phi^{s \rightarrow t}(f) = g$ , or

2. there is no element  $g \in \mathcal{B}[s \rightarrow t]$  that satisfies Equation 3.1 and  $\Phi^{s \rightarrow t}(f)$  is undefined.

Suppose that  $g$  and  $h$  are solutions to Equation 3.1. Then  $B^{s,t}(g, y) = B^{s,t}(h, y)$  for each  $y \in \mathcal{B}[s]$  since  $\Phi^s$  is a surjection. Extensionality therefore implies that  $g$  and  $h$  are equal. So, if there is a solution in  $\mathcal{B}[s \rightarrow t]$  for Equation 3.1, then there is a unique one.

The following is the basic fact about partial homomorphisms; it implies as a corollary the preservation of equations by partial homomorphisms.

**Lemma 3.4.2.** Let  $\mathcal{A}$  and  $\mathcal{B}$  be frames. If  $\Phi : \mathcal{A} \rightarrow \mathcal{B}$  is a partial homomorphism and  $\rho$  is an  $H$ -environment for  $\mathcal{A}$  and  $\rho'$  is an  $H$ -environment for  $\mathcal{B}$  such that  $\Phi^t(\rho(x)) = \rho'(x)$  for each variable  $x$  in  $H$ , then

$$\Phi^t(\mathcal{A}[H \triangleright M : t]\rho) = \mathcal{B}[H \triangleright M : t]\rho'$$

whenever  $H \vdash M : t$ .

**Corollary 3.4.3.** If there is a partial homomorphism  $\Phi : \mathcal{A} \rightarrow \mathcal{B}$  and  $\mathcal{A} \models (H \triangleright M = N : t)$ , then  $\mathcal{B} \models (H \triangleright M = N : t)$ .

**Proof.** Suppose  $\rho'$  is an  $H$ -environment for  $\mathcal{B}$ . Choose  $\rho$  so that  $\rho'(x) = \Phi^t(\rho(x))$  for each  $x$  in  $H$ . This is possible because  $\Phi^s$  is a surjection. Then  $\mathcal{B}[H \triangleright M : t]\rho' = \Phi^t(\mathcal{A}[H \triangleright M : t]\rho)$  and  $\Phi^t(\mathcal{A}[H \triangleright N : t]\rho) = \mathcal{B}[H \triangleright N : t]\rho'$  by Lemma 3.4.2. But  $\mathcal{A}[H \triangleright M : t]\rho$  and  $\mathcal{A}[H \triangleright N : t]\rho$  are equal by assumption. ■

**Lemma 3.4.4.** Let  $\mathcal{A}$  be a type frame and suppose there is a surjection from a set  $X$  onto  $\mathcal{A}[\mathbf{o}]$ . Then there is a partial homomorphism from  $\mathcal{F}_X$  (the full type frame over  $X$ ) to  $\mathcal{A}$ .

**Proof.** Let  $\Phi^{\circ} : X \rightarrow \mathcal{A}[\mathbf{o}]$  be any surjection. Suppose

$$\begin{aligned} \Phi^s &: \mathcal{F}_X[s] \rightarrow \mathcal{A}[s] \\ \Phi^t &: \mathcal{F}_X[t] \rightarrow \mathcal{A}[t] \end{aligned}$$

are partial surjections. We define  $\Phi^{s \rightarrow t}(f)$  to be the unique element of  $\mathcal{A}[s \rightarrow t]$ , if it exists, such that  $A^{s,t}(\Phi^{s \rightarrow t}(f), \Phi^s(y)) = \Phi^t(f(y))$  for all  $y$



in the domain of definition of  $\Phi^s$ . Proof that this defines a surjection is carried out by induction on structure of types. It holds by assumption for ground types; suppose  $g \in \mathcal{A}[[s \rightarrow t]]$  and  $\Phi^s, \Phi^t$  are surjections. Choose  $g' \in \mathcal{F}_X[[s \rightarrow t]] = \mathcal{F}_X[[s]] \rightarrow \mathcal{F}_X[[t]]$  such that, for all  $y$  in the domain of definition of  $\Phi^s$ , we have  $g'(y) \in (\Phi^t)^{-1}(A^{s,t}(g, \Phi^s(y)))$ . This is possible because  $\Phi^t$  is a surjection. Since  $\mathcal{A}$  is a type frame, extensionality implies that  $\Phi^{s \rightarrow t}(g') = g$ . By the definition of  $\Phi^s$  it is therefore a partial homomorphism. ■

**Theorem 3.4.5 (Completeness for full type frame).** *If  $X$  is infinite, then  $\vdash (H \triangleright M = N : t)$  if, and only if,  $\mathcal{F}_X \models (H \triangleright M = N : t)$ .*

**Proof.** We proved soundness ( $\Rightarrow$ ) earlier. To prove sufficiency ( $\Leftarrow$ ), begin by noting that Lemma 3.4.4 implies that there is a partial homomorphism from the full type frame,  $\mathcal{F}_X$ , onto the term model,  $\mathcal{T}$  where  $\mathcal{H}$  is chosen so that  $H \subseteq \mathcal{H}$ . If  $\mathcal{F}_X \models (H \triangleright M = N : t)$ , then  $\mathcal{T} \models (H \triangleright M = N : t)$  by Corollary 3.4.3. By Theorem 3.3.9, this means that  $\vdash (H \triangleright M = N : t)$ , the desired conclusion. ■

Partial homomorphisms are a special instance of a more general notion called a *logical relation* which serves as the one of the most basic tools for reasoning about types. Many of the properties of logical relations were developed by Tait, Statman, and Howard [Howard, 1973; Statman, 1982; Statman, 1985a; Statman, 1985b; Statman, 1986; Tait, 1967], and they continue to be a topic of interest for applications. A general survey on logical relations is included in [Mitchell, 1990], and [Burn *et al.*, 1986] furnishes an example of how logical relations can be applied to the static analysis of programs.

## 4 Simple types as domains

The simply-typed  $\lambda$ -calculus is too primitive to serve as a programming language: a programming language typically provides a notion of *evaluation* rather than simply an equational theory. Even when a directed use of equations for the simply-typed calculus is taken as an operational semantics, the resulting language is somewhat unexpressive. When evaluation is considered, the structures needed to model types must account for the semantics of such computational concepts as divergence and recursive definitions. This section considers how the theory of domains is related to the semantics of types for a basic language that extends the simply-typed calculus with a judicious collection of primitives in order to provide something like the power of a programming language.

### 4.1 A programming language for computable functions

The system known as PCF (Programming language for Computable Functions) was introduced by Dana Scott [Scott, 1969]. The variant of Scott's system

described here is taken from [Breazu-Tannen *et al.*, 1990]. Its types and terms are defined as follows:

$$\begin{aligned}
 t &::= \mathbf{num} \mid \mathbf{bool} \mid t \rightarrow t \\
 M &::= \mathbf{0} \mid \mathbf{true} \mid \mathbf{false} \mid \\
 &\quad \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{zero?}(M) \mid \mathbf{if } M \mathbf{ then } M \mathbf{ else } M \mid \\
 &\quad x \mid \lambda x : t. M \mid MM \mid \mu x : t. M
 \end{aligned}$$

The syntax of PCF essentially includes the terms of the simply-typed  $\lambda$ -calculus but with two ground types **num** and **bool**. Conventions for PCF syntax are similar to those for the basic simply-typed calculus.

Typing rules for PCF are those of the simply-typed  $\lambda$ -calculus (Table 4) together with those given in Table 6. Two basic facts about the type

**Table 6.** Typing rules for PCF

[Zero]	$H \vdash \mathbf{0} : \mathbf{num}$
[True]	$H \vdash \mathbf{true} : \mathbf{bool}$
[False]	$H \vdash \mathbf{false} : \mathbf{bool}$
[Pred]	$\frac{H \vdash M : \mathbf{num}}{H \vdash \mathbf{pred}(M) : \mathbf{num}}$
[Succ]	$\frac{H \vdash M : \mathbf{num}}{H \vdash \mathbf{succ}(M) : \mathbf{num}}$
[IsZero]	$\frac{H \vdash M : \mathbf{num}}{H \vdash \mathbf{zero?}(M) : \mathbf{bool}}$
[Cond]	$\frac{H \vdash L : \mathbf{bool} \quad H \vdash M : t \quad H \vdash N : t}{H \vdash \mathbf{if } L \mathbf{ then } M \mathbf{ else } N : t}$
[Rec]	$\frac{H, x : t \vdash M : t}{H \vdash \mu x : t. M : t}$

system are given by the following:

**Lemma 4.1.1.**

1. If  $H \vdash M : s$  and  $H \vdash M : t$ , then  $s \equiv t$ .
2. If  $H, x : s \vdash M : t$  and  $H \vdash N : s$ , then  $H \vdash [N/x]M : t$ .

Parts (1) and (2) are the analogs of Lemmas 3.1.2 and 3.1.4 respectively.

## 4.2 Operational semantics

To see PCF as a programming language we need to describe how its well-typed programs are evaluated. One approach to describing such a semantics is to indicate how a term  $M$  evaluates to another term  $M'$  by defining a relation  $M \rightarrow M'$  between closed terms using a set of *evaluation rules*. The goal of such rewriting is to obtain a *value* to which no further rules apply. In order to define precisely how a term is related to a value, we define a binary *transition relation*  $\rightarrow$  to be the least relation on pairs of PCF terms that satisfies the axioms and rules in Table 7. A set of evaluation rules

**Table 7.** Transition rules for call-by-name evaluation of PCF

---

$\frac{M \rightarrow N}{\text{pred}(M) \rightarrow \text{pred}(N)}$	$\text{pred}(0) \rightarrow 0 \quad \text{pred}(\text{succ}(V)) \rightarrow V$
$\frac{M \rightarrow N}{\text{zero?}(M) \rightarrow \text{zero?}(N)}$	
$\text{zero?}(0) \rightarrow \text{true} \quad \text{zero?}(\text{succ}(V)) \rightarrow \text{false}$	
$\frac{M \rightarrow N}{\text{succ}(M) \rightarrow \text{succ}(N)}$	
$\frac{M \rightarrow N}{M(L) \rightarrow N(L)}$	$(\lambda x : t. M)(N) \rightarrow [N/x]M$
$\text{if true then } M \text{ else } N \rightarrow M \quad \text{if false then } M \text{ else } N \rightarrow N$	
$\frac{L \rightarrow L'}{\text{if } L \text{ then } M \text{ else } N \rightarrow \text{if } L' \text{ then } M \text{ else } N}$	
$\mu x : t. M \rightarrow [\mu x : t. M/x]M$	

---

given in this form is sometimes called a *structural operational semantics (SOS)* because the hypotheses of the rules involve only the evaluation of proper structural components of the expressions in their conclusions. This approach to semantics was developed by Gordon Plotkin [Plotkin, 1976; Plotkin, 1981]. We say that a term  $M$  evaluates to a value  $V$  just in the case  $M \rightarrow^* V$  where  $\rightarrow^*$  is the transitive, reflexive closure of the transition

relation and a value is a term generated by the following grammar:

$$V ::= 0 \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{succ}(V) \mid \lambda x : t. M \quad (4.1)$$

Letters  $U, W$  range over values. The transition relation is deterministic:

**Lemma 4.2.1.** *If  $M \rightarrow N$  and  $M \rightarrow N'$  then  $N \equiv N'$ .*

One way to emphasize the structurality of the rules in Table 7 is to represent them using a grammar. An *evaluation context* for PCF is described by the following grammar:

$$E ::= [] \mid \mathbf{pred}(E) \mid \mathbf{zero?}(E) \mid \mathbf{succ}(E) \mid E(L) \mid \mathbf{if } E \mathbf{ then } M \mathbf{ else } N$$

where  $[]$  is intended to represent a ‘hole’ in a PCF term. A term of PCF is obtained from an evaluation context by filling the ‘hole’ in the context by a term: this is written in the form  $E[M]$ . (An examination of the grammar reveals that a context  $E$  has exactly one ‘hole’ in it.) If we now take analogs of the *axioms* (as opposed to the *rules*) from Table 7:

$$\begin{aligned} \mathbf{pred}(0) &\Rightarrow 0 \\ \mathbf{pred}(\mathbf{succ}(V)) &\Rightarrow V \\ \mathbf{zero?}(0) &\Rightarrow \mathbf{true} \\ \mathbf{zero?}(\mathbf{succ}(V)) &\Rightarrow \mathbf{false} \\ (\lambda x : t. M)(N) &\Rightarrow [N/x]M \\ \mathbf{if } \mathbf{true} \mathbf{ then } M \mathbf{ else } N &\Rightarrow M \\ \mathbf{if } \mathbf{false} \mathbf{ then } M \mathbf{ else } N &\Rightarrow N \end{aligned}$$

then the desired relation is defined by using the following rule:

$$\frac{M \Rightarrow N}{E[M] \rightarrow E[N]}.$$

This approach to describing a structural operational semantics was introduced in [Felleisen and Friedman, 1986].

There are other approaches to describing the evaluation of a programming language. One idea is to describe the relation  $M \rightarrow^* V$  more directly using a new set of rules. Such a description is sometimes known as a *natural (operational) semantics* because a semantics given in this form resembles a natural deduction system for a logic. An early instance of such a semantics appears in [Martin-Löf, 1971] and more recent examples in [Clément *et al.*, 1986; Kahn, 1987] and the Standard for ML [Milner *et al.*, 1990; Milner and Tofte, 1991]; a comparative discussion can be found in [Gunter, 1993].

Let us now look at such a semantics for PCF. It is given by a binary relation  $\Downarrow$  between closed terms of the calculus. The binary relation is

defined as the least relation that satisfies the axioms and rules in Table 8. In the description of these rules, the terms that appear on the right side have been written using the letters  $U, V, W$  for values rather than  $L, M, N$  for arbitrary terms. To read the rules, assume at first that  $U, V, W$  range over all terms. It can then be proved by an induction on the height of a derivation that if  $M \Downarrow V$  for any terms  $M$  and  $V$ , then  $V$  is a term generated by the grammar 4.1. In other words, if rules such as

$$\frac{M \Downarrow \text{succ}(V)}{\text{pred}(M) \Downarrow V}$$

were instead written in the form

$$\frac{M \Downarrow \text{succ}(N)}{\text{pred}(M) \Downarrow N}$$

then it would be possible to *prove* that  $N$  has the form of a value  $V$ . It

Table 8. Natural rules for call-by-name evaluation of PCF

$0 \Downarrow 0$	$\text{true} \Downarrow \text{true}$	$\text{false} \Downarrow \text{false}$
$\frac{M \Downarrow 0}{\text{pred}(M) \Downarrow 0}$	$\frac{M \Downarrow \text{succ}(V)}{\text{pred}(M) \Downarrow V}$	$\frac{M \Downarrow V}{\text{succ}(M) \Downarrow \text{succ}(V)}$
$\frac{M \Downarrow 0}{\text{zero?}(M) \Downarrow \text{true}}$	$\frac{M \Downarrow \text{succ}(V)}{\text{zero?}(M) \Downarrow \text{false}}$	
$\lambda x : s. M \Downarrow \lambda x : s. M$	$\frac{M \Downarrow \lambda x : s. M' \quad [N/x]M' \Downarrow V}{M(N) \Downarrow V}$	
$\frac{M_1 \Downarrow \text{true} \quad M_2 \Downarrow V}{\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \Downarrow V}$	$\frac{M_1 \Downarrow \text{false} \quad M_3 \Downarrow V}{\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \Downarrow V}$	
$\frac{[\mu x : t. M/x]M \Downarrow V}{\mu x : t. M \Downarrow V}$		

is not hard to check that if  $M \Downarrow U$  and  $M \Downarrow V$ , then  $U \equiv V$ , so  $\Downarrow$  is a partial function. It is, moreover, possible to show that this gives the same semantics for PCF as the SOS:



**Theorem 4.2.2.**  $M \Downarrow V$  if, and only if,  $M \rightarrow^* V$ .

### 4.3 Operational equivalence

The question, now, is what these various formulations of the operational semantics of PCF have to do with the *types* for the system. Let us assume the perspective of the previous section and consider the question of what the interpretations of the PCF types should be. Interpreting them as sets as we did for the simply-typed  $\lambda$ -calculus leads to problems, however, when we wish to interpret recursion. The best-known approach to resolving this difficulty is to impose additional structure on the interpretations of types by using certain kinds of ordered sets, ordinarily known as *domains*. Assuming that we have found such a semantics—one that interprets a term  $M$  of type  $t$  as an element of the interpretation of  $t$  (modulo the values of free variables of  $M$ )—the key issue is the relationship between this form of semantics and the operational semantics of the language as described above. That is, we need the analogs to the soundness and completeness theorems given earlier, but now these results should be relative to an operational semantics rather than an equational theory. The trick is, in effect, to generate an equational theory from the operational semantics and study these properties relative to it. More precisely, this is done relative to a pre-order imposed on terms; this pre-order induces the desired equations.

A PCF *context*  $C$  is essentially a PCF term with a missing subterm marked by a place-holder  $[]$ . The PCF term obtained by filling the ‘hole’ in the term  $C$  by a term  $M$  is denoted  $C[M]$ . This is similar to substitution, but the placement of  $M$  into context  $C$  permits free variables of  $M$  to be bound within variable scopes determined by  $C$ . (In particular, contexts, unlike terms, are not considered equivalent modulo renaming of bound variables.) Evaluation contexts are a special class of contexts in which the missing subterm is in a special position. We define the key notion of operational equivalence as follows. Suppose  $M$  and  $N$  are terms of type  $t$  (that is,  $H \vdash M : t$  and  $H \vdash N : t$  for some  $H$ ). Say  $M$  is an operational approximation of  $N$  and write  $M \sqsubseteq_o N$  if, for every context  $C$  such that  $C[M]$  and  $C[N]$  are closed terms of ground type,

$$C[M] \Downarrow V \text{ implies } C[N] \Downarrow V.$$

It is possible to prove that  $\sqsubseteq_o$  is a pre-order; terms  $M, N$  are *operationally equivalent*, and we write  $M \approx N$  if  $M \sqsubseteq_o N$  and  $N \sqsubseteq_o M$ .

A semantics  $\llbracket \cdot \rrbracket$  is said to be *adequate* if  $\llbracket H \triangleright M : t \rrbracket = \llbracket H \triangleright N : t \rrbracket$  implies  $N \approx M$ . It is said to be *fully abstract* if it is sound and  $N \approx M$  implies  $\llbracket H \triangleright M : t \rrbracket = \llbracket H \triangleright N : t \rrbracket$ . Two well-known adequate semantics for PCF are the *bc-domains* interpretation  $\mathcal{C}[\cdot]$  and the *dI-domains* interpretation  $\mathcal{D}[\cdot]$ . To describe each of these briefly some knowledge of domain theory will be assumed: some definitions are given below—further

background can be found in [Abramsky and Jung, 1994].

#### 4.4 bc-domains and dI-domains

A *bc-domain* is an algebraic complete partial order that is bounded complete, that is, every subset that has an upper bound has a least upper bound (lub). If  $D$  and  $E$  are bc-domains, then the space of continuous functions  $[D \rightarrow E]$  under the pointwise order is also a bc-domain. Such domains can be used to model PCF types by interpreting ground type expressions **num** and **bool** as the flat cpo's  $\mathbb{N}_\perp$  (numbers together with least element  $\perp$ ) and  $\mathbb{T}$  (truth values **true**, **false** together with  $\perp$ ) respectively and the higher types by taking  $C[s \rightarrow t]$  to be  $[C[s] \rightarrow C[t]]$ . As before, meanings are defined on triples  $H, M, t$  where  $H \vdash M : t$ . An  $H$ -environment  $\rho$  is a partial function that assigns to each variable  $x$  such that  $x \in H$  a value  $\rho(x)$  in  $C[H(x)]$ . The meaning  $C[H \triangleright M : t]$  is a function that assigns to each  $H$ -environment  $\rho$  a value

$$C[H \triangleright M : t]\rho \in C[t].$$

The definition of this function follows the structure of the expression  $M$  (or, equivalently, the proof that  $H \vdash M : t$ ). For the simply-typed  $\lambda$ -calculus fragment of PCF, the interpretation looks the same as before. The arithmetic and conditional expressions have a straightforward interpretation. It is the interpretation of recursive functions that takes advantage of the additional structure of *domains*—as compared to *sets*—in the interpretation of types:

$$C[H \triangleright \mu x : t. M : t]\rho = \text{fix}(d \mapsto C[H, x : t \triangleright M : t]\rho[x \mapsto d])$$

where **fix** is a function that gives the least fixed point of a continuous function. To show that the definition makes sense, one proves the following:

**Lemma 4.4.1.** *If  $H' = H, x : s$  is a type assignment such that  $H' \vdash M : t$ , then the function*

$$d \mapsto C[H' \triangleright M : t]\rho[x \mapsto d]$$

*is continuous for any  $H'$ -environment  $\rho$ .*

The following property is also easy to establish by induction on the height of a derivation tree:

**Proposition 4.4.2.** *If  $M \rightarrow N$ , then  $C[M] = C[N]$ .*

Proving that adequacy holds for  $C[\cdot]$  is somewhat harder and beyond the scope of this chapter.

Another interpretation of PCF can be obtained by using bc-domains that are distributive and have property I. To define these properties precisely, let  $\sqcup$  and  $\sqcap$  stand for the least upper bound and greatest lower bound operators respectively.

**Definition 4.4.3.** A bc-domain  $D$  is said to be *distributive* if  $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$  whenever  $\{x, y\}$  has an upper bound. An algebraic cpo  $D$  has *property I* if  $\{x \mid x \sqsubseteq a\}$  is finite for each compact element  $a$  of  $D$ . A distributive bc-domain that satisfies property I is called a *dI-domain*.

Interpreting the types of PCF as dI-domains requires one more crucial idea. It is not hard to find dI-domains  $D, E$  with the property that the continuous function space  $[D \rightarrow E]$  under the pointwise order is *not* a dI-domain. Higher types for a model based on dI-domains cannot be interpreted with this construct. The key idea is given in the following:

**Definition 4.4.4.** A continuous function  $f : D \rightarrow E$  between dI-domains  $D$  and  $E$  is *stable* if  $f(x \sqcap y) = f(x) \sqcap f(y)$  whenever  $\{x, y\}$  has an upper bound. If  $f, g : D \rightarrow E$  are stable, then  $f$  is below  $g$  in the *stable ordering* and we write  $f \sqsubseteq_s g$  if

$$x \sqsubseteq y \text{ implies } f(x) = f(y) \sqcap g(x)$$

for each  $x, y \in D$ .

It is possible to show that if  $D, E$  are dI-domains, then the poset of stable functions  $[D \rightarrow_s E]$  under the stable ordering is also a dI-domain. The dI-domains can be used to give a semantics  $\mathcal{D}[\cdot]$  for PCF in basically the same way that bc-domains were used to give a semantics  $\mathcal{C}[\cdot]$  before. One must prove the analog of Lemma 4.4.1:

**Lemma 4.4.5.** *If  $H, x : s \vdash M : t$ , then the function*

$$d \mapsto \mathcal{D}[H, x : s \triangleright M : t]\rho[x \mapsto d]$$

*is stable.*

## 4.5 Full abstraction

The use of algebraic cpo's and bounded completeness as a model of  $\lambda$ -calculus was developed by Dana Scott [1976; 1981; 1982a; 1982b] and [Gunter and Scott, 1990]. The dI-domains were introduced by Gerard Berry [1978; 1979], [Berry *et al.*, 1985] in an effort to find a fully abstract model of PCF after Gordon Plotkin [1976] demonstrated that the bc-domains model of PCF is *not* fully abstract. Plotkin proved this failure directly from the operational semantics of PCF, but the result can also be obtained semantically by using dI-domains as a 'non-standard' model. To do this, we need to demonstrate two terms that have the same operational behaviour in all ground contexts but fail to be equal in the model. To this end, let

$$T, F : (\text{bool} \rightarrow (\text{bool} \rightarrow \text{bool})) \rightarrow \text{bool}$$

be the PCF terms given in Table 9. In the table there,  $\Omega$  is a divergent

**Table 9.** Operationally equivalent programs with different denotations

---

$T \equiv \lambda f : \mathbf{bool} \rightarrow (\mathbf{bool} \rightarrow \mathbf{bool}).$ <b>if</b> $f(\mathbf{true})(\Omega)$ <b>then</b> <b>if</b> $f(\Omega)(\mathbf{true})$ <b>then</b> <b>if</b> $f(\mathbf{false})(\mathbf{false})$ <b>then</b> $\Omega$ <b>else</b> $\mathbf{true}$ <b>else</b> $\Omega$ <b>else</b> $\Omega$
$F \equiv \lambda f : \mathbf{bool} \rightarrow (\mathbf{bool} \rightarrow \mathbf{bool}).$ <b>if</b> $f(\mathbf{true})(\Omega)$ <b>then</b> <b>if</b> $f(\Omega)(\mathbf{true})$ <b>then</b> <b>if</b> $f(\mathbf{false})(\mathbf{false})$ <b>then</b> $\Omega$ <b>else</b> $\mathbf{false}$ <b>else</b> $\Omega$ <b>else</b> $\Omega$

---

program of boolean type (for instance  $\mu x : \mathbf{bool}. x$  will do). The term  $F$  is the same as  $T$  except for the occurrence **false** in the fifth line.

Now, the programs  $T$  and  $F$  have the same operational behaviour in all ground contexts. To see this, it suffices, by adequacy for the dI-domains model, to show that  $\mathcal{D}[T] = \mathcal{D}[F]$ . We show, in fact, that  $\mathcal{D}[T](\emptyset) = \mathcal{D}[F](\emptyset) : f \mapsto \perp$ , where  $\emptyset$  is the ‘arid’ environment (which makes no assignments). To this end, suppose  $\mathcal{D}[T](\emptyset)(f) \neq \perp$ . This can only happen if  $\mathcal{D}[T](\emptyset)(f) = \mathbf{true}$ . If  $f'(x, y) = f(x)(y)$  is the ‘uncurrying’ of  $f$ , then

$$\begin{aligned} f(\perp)(\perp) &= f'(\perp, \perp) = f'((\mathbf{true}, \perp) \sqcap (\perp, \mathbf{true})) \\ &= f'(\mathbf{true}, \perp) \sqcap f'(\perp, \mathbf{true}) = \mathbf{true} \end{aligned}$$

since  $f'$  is stable. On the other hand,  $f(\mathbf{false})(\mathbf{false}) = f'(\mathbf{false}, \mathbf{false}) = \mathbf{false}$ , and this contradicts the monotonicity of  $f'$ . A similar argument applies to  $F$ , so we can conclude that the terms  $T$  and  $F$  have the same operational behaviour. Indeed, they both have the same operational behaviour as  $\lambda x. \Omega$ . Switching now to the interpretations of these terms in the bc-domain semantics, we can show that  $\mathcal{C}[T] \neq \mathcal{C}[F]$ . To do this, consider a function **por** :  $\mathbf{T} \rightarrow [\mathbf{T} \rightarrow \mathbf{T}]$ , called the *parallel or*, defined by the left truth table in Table 10 where the values in the left column are those of the first argument and the values in the top row are those of the second argument. This can be contrasted with the truth table for the (left-to-right) sequential or defined by

$$\mathbf{or} = \mathcal{C}[\lambda x : \mathbf{bool}. \lambda y : \mathbf{bool}. \mathbf{if } x \mathbf{ then true else } y](\emptyset).$$

**Table 10.** Truth tables for parallel and sequential disjunction

---

<b>por</b>	<b>true</b>	<b>false</b>	$\perp$
<b>true</b>	<b>true</b>	<b>true</b>	<b>true</b>
<b>false</b>	<b>true</b>	<b>false</b>	$\perp$
$\perp$	<b>true</b>	$\perp$	$\perp$
<b>or</b>	<b>true</b>	<b>false</b>	$\perp$
<b>true</b>	<b>true</b>	<b>true</b>	<b>true</b>
<b>false</b>	<b>true</b>	<b>false</b>	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

---

Note the difference in the value of **or**( $\perp$ )(**true**) in the truth table for **or** given in Table 10. Now, the function **por** is monotone on a finite domain and therefore continuous, so it is an element of the interpretation  $\mathcal{C}[\mathbf{bool} \rightarrow (\mathbf{bool} \rightarrow \mathbf{bool})]$ . Hence

$$\mathcal{C}[T](\emptyset)(\mathbf{por}) = \mathbf{true} \neq \mathbf{false} = \mathcal{C}[F](\emptyset)(\mathbf{por})$$

so  $\mathcal{C}[T] \neq \mathcal{C}[F]$ , and  $\mathcal{C}[\cdot]$  is therefore not fully abstract.

It was shown by Berry that the dI-domains are *also* not fully abstract. Here is a brief semantic proof using the bc-domains model as a ‘non-standard’ interpretation. We show that two terms with different meanings in the dI-domains model have the same operational behaviour. To this end, define monotone functions  $p, q$  on the truth value poset by taking  $p : x \mapsto \mathbf{true}$  and taking  $q$  to be the function

$$q(x) = \begin{cases} \perp & \text{if } x = \perp \\ \mathbf{true} & \text{otherwise.} \end{cases}$$

The function  $q$  is below  $p$  in the pointwise order, but these two functions are unrelated in the stable order. Noting this, it is possible to see that the following function is an element of  $\mathcal{D}[(\mathbf{bool} \rightarrow \mathbf{bool}) \rightarrow \mathbf{bool}]$ :

$$r(x) = \begin{cases} \mathbf{true} & \text{if } x = q \\ \mathbf{false} & \text{if } x = p \\ \perp & \text{otherwise.} \end{cases}$$

Now consider the programs

$$\begin{aligned} M &\equiv \lambda f. \text{if } f(q) \text{ then (if } f(p) \text{ then } \Omega \text{ else true) else } \Omega \\ N &\equiv \lambda f. \text{if } f(q) \text{ then (if } f(p) \text{ then } \Omega \text{ else false) else } \Omega \end{aligned}$$

where type tags have been dropped to reduce clutter. Clearly  $\mathcal{D}[M](r) \neq$



$\mathcal{D}[N](r)$ . However,  $M$  and  $N$  have the same operational behaviour because  $\mathcal{C}[M] = \mathcal{C}[N]$ . To see why this latter equation holds, just note that if  $f(q) = \mathbf{true}$  for any function  $f$  that is monotone over the pointwise-ordered monotone functions of type  $\mathbf{bool} \rightarrow \mathbf{bool}$ , then  $f(p) = \mathbf{true}$  as well.

The problem of finding a fully abstract model of PCF has a long history. But rather than ask whether PCF has such a model, one can also ask whether there is any extension of PCF that is fully abstract with respect, say, to the bc-domains semantics. Plotkin [Plotkin, 1976] showed that this is the case; indeed, the language PCF is fully abstract for the bc-domains model if one adds a primitive for computing the parallel disjunction **por** [Stoughton, 1991]. Interestingly, no similar way of extending the language seems to work for the dI-domains model [Jim and Meyer, 1991]. More details on full abstraction and related topics can be found in [Gunter, 1992]. Some current research directions are indicated in the concluding section of this chapter.

## 5 Types as invariants

In Section 4 the idea of presenting an operational semantics for PCF was described. The semantics of types was given in terms of domains and then this interpretation was related to the equivalence induced by the operational semantics. But there is another basic relationship that one can show between the operational semantics and the types of the language based on the simple idea that the types are *invariants* of the operational semantics. This is a property known as *subject reduction*. For PCF and the SOS given in Table 7 it can be expressed as follows:

**Theorem 5.0.1 (Subject reduction).** *If  $H \vdash M : t$  and  $M \rightarrow N$ , then  $H \vdash N : t$ .*

By Theorem 4.2.2 this has the following corollary:

**Corollary 5.0.2.** *If  $H \vdash M : t$  and  $M \Downarrow N$ , then  $H \vdash N : t$ .*

This form of theorem is very useful for proving that certain kinds of run-time errors are avoided by programs that are type correct.

### 5.1 Run-time safety

What properties are expected for the evaluation of a type-correct program beyond those that may hold of an arbitrary one? To appreciate the significance of the types, look again at the operational rules in Table 8. Take a typical rule such as the one for application in call-by-name:

$$\frac{M \Downarrow \lambda x : t. M' \quad [N/x]M' \Downarrow V}{M(N) \Downarrow V}$$

This is the only rule whose conclusion describes how to derive a value for an application, so any attempt to prove that  $M(N)$  has value  $V$  must use

it. The rule requires that two hypotheses be established. Let us focus on the first. It was remarked before that if  $M \Downarrow U$  for some value  $U$ , then  $U$  is the unique value that satisfies this relationship. Hence there are three possibilities that could result from the attempt to find a value for  $M$ :

1. there is no value  $U$  such that  $M \Downarrow U$ , or
2. there is a term  $M'$  such that  $M \Downarrow \lambda x : t. M'$ , or
3. there is a term  $U$  such that  $M \Downarrow U$ , but  $U$  does not have the form  $\lambda x : t. M'$ .

The first of these might occur because of divergence, or perhaps for some other reason. The second is the conclusion we must reach to find a value for  $M(N)$ . The third case arises in an ‘abnormal’ situation in which something other than an abstraction is being applied to an argument. For example, this would happen if we attempted to evaluate the application  $\mathbf{0}(\mathbf{0})$  of the number  $\mathbf{0}$  to itself. Here is what the type-correctness of  $M(N)$  ensures: the third possibility above never occurs. In the example  $\mathbf{0}(\mathbf{0})$  this is clear because  $\mathbf{0}$  has type **num** rather than type **num**  $\rightarrow t$  as it would be required to have if it is to be applied to a number.

Although the first and third cases above both mean that  $M(N)$  does not have a value, there is an important difference in the way this failure occurs. In particular, if it is found that  $M \Downarrow U$  but  $U$  does not have the desired form, then it is possible to report immediately that the attempt to find a value for  $M(N)$  has *failed*. This will not always be possible for the first case, since the failure to find a value for  $M$  may be due to an infinite regression of attempts to apply operational rules (this is what would happen for the term  $\mu x : \mathbf{num}. x$ , for example). Any attempt to determine whether this is the case through an effective procedure will fail, because this is tantamount to solving the halting problem. Hence, the last case is special.

For some guidance, let us consider the difference between these possibilities in a programming language. Here is an example of a Scheme program that will diverge when applied to an argument:

```
(define (f x) (f x))
```

Evaluating  $(f\ 0)$  in the read-eval-print loop will be a boring and unfulfilling activity that will probably be ended by an interruption by the programmer. This program diverges and therefore does not have a value. On the other hand, what happens if we attempt to evaluate the program  $(0\ 0)$  in the read-eval-print loop? There is no value for this expression, but we receive an instant warning of this limitation that may look like this:

#### Application of inapplicable object 0

The difference between these two outcomes arises from the distinction between *divergence* and a *run-time type error*. While divergence is generally undetectable, the run-time type error can be reported when it arises.

To study these ideas rigorously, let us focus on a specific language. The following grammar defines the syntax of type expressions  $t$  and terms  $M$  of a calculus called PCF *with type errors*. The extended calculus is the same as PCF except for the inclusion of a new constant called **tyerr**. Here is the expanded grammar:

$$\begin{aligned}
 t &::= \mathbf{num} \mid \mathbf{bool} \mid t \rightarrow t \\
 M &::= \mathbf{tyerr} \mid \mathbf{0} \mid \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{zero?}(M) \mid \\
 &\quad x \mid \lambda x : t. M \mid MM \mid \mu x : t. M \mid \mathbf{if } M \mathbf{ then } M \mathbf{ else } M
 \end{aligned}$$

The typing rules for extended PCF are the same as those for PCF itself. Note, in particular, that the relation  $H \vdash \mathbf{tyerr} : t$  fails for each  $H$  and  $t$  since there is no typing rule for proving such a relation.

The rules for a natural operational semantics are those given earlier in Table 8 together with the ‘error’ rules given in Table 11. Values in the new

**Table 11.** Operational rules for type errors

$\mathbf{tyerr} \Downarrow \mathbf{tyerr}$		$\frac{L \Downarrow V \quad V \notin \text{Boolean}}{\mathbf{if } L \mathbf{ then } M \mathbf{ else } N \Downarrow \mathbf{tyerr}}$	
$M \Downarrow V \quad V \notin \text{Number}$	$\mathbf{pred}(M) \Downarrow \mathbf{tyerr}$	$M \Downarrow V \quad V \notin \text{Number}$	$\mathbf{succ}(M) \Downarrow \mathbf{tyerr}$
$M \Downarrow V \quad V \notin \text{Number}$	$\mathbf{zero?}(M) \Downarrow \mathbf{tyerr}$	$M \Downarrow V \quad V \notin \text{Lambda}$	$M(N) \Downarrow \mathbf{tyerr}$

language are those of PCF together with the term **tyerr** for a type error. The rules in the table are defined using syntactic judgements such as  $V \notin \text{Boolean}$  where **Boolean** is the set of values  $V$  such that  $\vdash V : \text{Boolean}$ . In the rules, **Number** is the set of values of numerical type (that is, numerals) and **Lambda** those of higher type (that is, abstractions). The expanded set of rules has properties similar to those of the system without explicit error elements. For example, the relation  $\Downarrow$  for the full language is still a partial function, that is, if  $M \Downarrow U$  and  $M \Downarrow V$ , then  $U \equiv V$ . What differentiates this system from the previous one is the fact that the hypotheses of the rules now cover all possible patterns for the outcome of an evaluation. If the evaluation of a term calls for the evaluation of other terms, then the error rules indicate what is to be done if the result of evaluating these other terms is a type error or yields a conclusion having the wrong form.

Now, we would like to prove a theorem that says that the evaluation of a well-typed term does not produce a type error. Here is a more precise statement:

**Theorem 5.1.1.** *If  $H \vdash M : t$  then it is not the case that  $M \Downarrow \mathbf{tyerr}$ .*

The result we need to show absence of runtime type errors can be proved using a form of subject reduction theorem. In this case the result must be proved relative to the typing system for PCF and the evaluation relation  $\Downarrow$  defined as the least relation satisfying the rules in Table 11 as well as those in Table 8.

**Theorem 5.1.2 (Subject reduction).** *Let  $M$  be a term in extended PCF. If  $M \Downarrow V$  and  $H \vdash M : t$ , then  $H \vdash V : t$ .*

**Proof.** The proof is by induction on the height of the evaluation  $M \Downarrow V$ . I will do only the cases for predecessor, application, and recursion.

Case  $M \equiv \mathbf{pred}(M')$ . There are three possibilities for the last step in the derivation of  $M \Downarrow V$ . If  $M' \Downarrow V'$ , then we employ the induction hypothesis to conclude that  $H \vdash V' : \mathbf{num}$ . In particular, this means that  $V' \in \text{Number}$  so  $V' \equiv \mathbf{0}$  or the last step of the derivation must be an application of the rule

$$\frac{M \Downarrow \mathbf{succ}(V) \quad V \in \text{Number}}{\mathbf{pred}(M) \Downarrow V}$$

where  $V' \equiv \mathbf{succ}(V)$ . If  $V' \equiv \mathbf{0}$ , then the desired conclusion is immediate, since  $H \vdash \mathbf{0} : \mathbf{num}$ . On the other hand, the only way to have  $H \vdash \mathbf{succ}(V) : \mathbf{num}$  is if  $H \vdash V : \mathbf{num}$ , so this possibility also leads to the desired conclusion.

Case  $M \equiv L(N)$ . Say  $H \vdash L : r \rightarrow s$  and  $H \vdash N : r$ . There are two operational rules that may apply to the evaluation of an application. The error rule in Table 7.3 could not apply to  $M$ , however, because of the inductive hypotheses on  $L$ . Hence the last step in the evaluation of  $M$  must have the following form:

$$\frac{L \Downarrow \lambda x : r. L' \quad [N/x]L' \Downarrow V}{L(N) \Downarrow V}$$

Now, by the induction hypothesis,  $H \vdash \lambda x : r. L' : r \rightarrow s$  so it must be that  $H, x : r \vdash L' : s$ . Hence, by Lemma 4.1.1(2),  $H \vdash [N/x]L' : s$ , and it therefore follows from the induction hypothesis that  $H \vdash V : t$ .

Case  $M \equiv \mu x : t. M'$ . In this case,  $[\mu x : t. M'/x]M' \Downarrow V$ . By Lemma 4.1.1,  $H \vdash [\mu x : t. M'/x]M' : t$  so  $H \vdash V : t$  by the induction hypothesis. ■

Theorem 5.1.1 now follows immediately, since  $\mathbf{tyerr}$  does not have a type.

The interest of Theorem 5.1.1, which is intended to assert that the evaluation of a well-typed program does not yield a type error, depends entirely on the nature of the error rules. Hence it is important to examine them closely to see that they do indeed encode all of the circumstances



under which one would expect a type error to be reported. One problem with this way of asserting freedom from run-time errors is that mistakenly omitting a rule from Table 11 would make Theorem 5.1.1 easier to prove! Another way to express freedom from run-time errors is to use an SOS for the language and assert the result as a guarantee that computation does not get ‘stuck’ at a non-value. Here is such an assertion for PCF:

**Theorem 5.1.3.** *If  $H \vdash M : t$  and  $M$  is not a value, then  $M \rightarrow N$  for some  $N$ .*

## 5.2 Implicit types

If having a type is viewed primarily as a property of a program that ensures desirable runtime behaviour, then it may be a convenience if type-correctness is inferred automatically and programmers are relieved as far as possible of the need to write type annotations. This leads us to the idea of an *implicit* type, that is, one not explicitly given as part of the program. Since a program without explicit types naturally provides less type information than one that has them, a key technical issue arises for such programs: while our discussion has been focused entirely on languages for which the type of a term, if it has one, is uniquely determined because of type tags, this property must be viewed differently when type tags are omitted. Let us now consider the  $\lambda$ -calculus *without* the type tags. Since the tags gave the types explicitly before, the new system is called the *implicitly-typed* (simply-typed) calculus. The syntax for the language is simply

$$M ::= x \mid \lambda x. M \mid MM$$

and the various syntactic conventions are exactly the ones used earlier for the simply-typed  $\lambda$ -calculus with type tags. The typing rules for the implicit system are almost the same as those for the explicit calculus, but the abstraction rule now has different properties. The rules are given in Table 12.

**Table 12.** Implicitly-typed  $\lambda$ -calculus

---

[Proj]	$\frac{x \in H}{H \vdash x : H(x)}$
[Abs] <sup>−</sup>	$\frac{H, x : s \vdash M : t}{H \vdash \lambda x. M : s \rightarrow t}$
[Appl]	$\frac{H \vdash M : s \rightarrow t \quad H \vdash N : s}{H \vdash M(N) : t}$

---

Let us consider now how one could find a type for a term with respect



to the rules in Table 12, recalling that it was shown earlier that  $[\text{Abs}]^-$  leads to the failure of Lemma 3.1.2. If  $M$  is a term of the form  $L(N)$ , then types need to be found for  $L$  and  $N$ . If  $M$  has the form  $\lambda x. M'$ , then a type needs to be found for  $M'$  assuming that  $x$  has *some* type  $s$ . Now, in the case that we are looking for the type of a term like  $\lambda x. x$ , then, for *any* type  $s$ , we can find a type with the following instances of the projection and application rule:

$$\frac{x : s \vdash x : s}{\vdash \lambda x. x : s \rightarrow s}$$

The choice of  $s$  here is arbitrary, but each of the types for  $\lambda x. x$  must have the form  $s \rightarrow s$ . Indeed this form precisely characterizes what types *can* be the type of  $\lambda x. x$ . Let us now consider a slightly more interesting example; let

$$M \equiv \lambda x. \lambda f. f(x).$$

In a typing derivation ending with a type for  $M$ , the last two steps must have the following form:

$$\frac{\frac{x : t_1, f : t_2 \vdash f(x) : t_3}{x : t_1 \vdash \lambda f. f(x) : t_2 \rightarrow t_3}}{\vdash M : t_1 \rightarrow (t_2 \rightarrow t_3)}$$

Letting  $H$  be the assignment  $x : t_1, f : t_2$ , the derivation of the hypothesis at the top must have the form

$$\frac{H \vdash x : t_1 \quad H \vdash f : t_2}{x : t_1, f : t_2 \vdash f(x) : t_3}$$

where, to match the application rule, it must be the case that  $t_2$  have the form  $t_1 \rightarrow t_3$ . It is not hard to see that *any* choice of  $t_1, t_2, t_3$  satisfying this one condition will be derivable as a type for  $M$ . In short, the types that  $M$  can have are exactly characterized as those of the form

$$r \rightarrow (r \rightarrow s) \rightarrow s. \tag{5.1}$$

This suggests that there may be a way to reconstruct a general form for the type of a term in the implicit calculus. If the types  $r$  and  $s$  could be viewed as variables in the type (5.1), then we could say that a type  $t$  satisfies  $\vdash M : t$  just in case  $t$  is a substitution instance of (5.1). What is most important though is the prospect that a type for a term could be determined even though the type tags are missing. For a calculus like PCF, this might make it possible to omit type tags and still ensure the kind of security asserted for the well-typed terms of the language in Theorem 5.1.1.

Of course, there are terms that cannot be given a type because they ‘make no sense’. This is easy to see in PCF where there are constants:

a term such as  $\mathbf{0}(\mathbf{0})$  is clearly meaningless. In the  $\lambda$ -calculus by itself, however, there is a gray area between what is and what is not a sensible term. The implicit system of Table 12 judges that  $\vdash M : t$  if, and only if, there is a term of the *explicitly*-typed  $\lambda$ -calculus of Table 4 from which  $M$  can be obtained by erasing the tags. For example, it is impossible to find a type for  $\lambda f. f(f)$  with the implicit typing system. To see this, suppose on the contrary that this term does have a type. The derivation of the type must end with an instance of  $[\text{Abs}]^-$ :

$$\frac{f : s \vdash f(f) : t}{\vdash \lambda f. f(f) : s \rightarrow t}$$

The proof of the hypothesis must be an instance of  $[\text{Appl}]$ :

$$\frac{f : s \vdash f : u \rightarrow t \quad f : s \vdash f : u}{f : s \vdash f(f) : t},$$

which, by the axiom  $[\text{Proj}]$ , means that  $u \rightarrow t \equiv s \equiv u$ . However, there is no type that has this property since the type  $u$  cannot have itself as a proper subterm. But there are contexts in which this term seems to make some sense. For example, it might be argued that the term  $(\lambda x. x(x))(\lambda y. y)$  is harmless, since the  $x$  in the first abstraction is bound to an argument in the application that can indeed be applied to itself.

The calculus  $\text{ML}_0$ , which we now introduce, has a type system that can be viewed as a compromise between the implicit type discipline of Table 12 (which is essentially the simply-typed  $\lambda$ -calculus) and the *untyped*  $\lambda$ -calculus for which no typing system is used.  $\text{ML}_0$  is a core representation of the system of the ML programming language. The goal is to provide some level of additional flexibility to the implicit typing discipline while maintaining a close link to the simply-typed  $\lambda$ -calculus. The key idea in the system is the inclusion of a syntax class of parameterized types. The full grammar for the language is given as follows:

$$\begin{array}{ll} x & \in \text{TermVariable} \\ a & \in \text{TypeVariable} \\ t & ::= a \mid t \rightarrow t \\ T & ::= t \mid \Pi a. T \\ M & ::= x \mid \lambda x. M \mid MM \mid \text{let } x = M \text{ in } M \end{array}$$

In addition to the primitive syntax class of term variables  $x$ , a new syntax class of *type variables*  $a$  has been added. A *type scheme*  $T$  has the form  $\Pi a_1. \Pi a_2. \dots \Pi a_n. t$  where the type variables  $a_1, \dots, a_n$  bind any occurrences of these variables in the type  $t$ . The usual rules for substitution and  $\alpha$ -equivalence apply to type schemes: expressions are taken modulo  $\alpha$ -equivalence, and a substitution should not result in any free variable of the

substituted type becoming bound after the substitution. In other words, for any substitution  $\sigma$ ,

$$\sigma(\Pi a_1 \dots \Pi a_n. t) \equiv \Pi a_1 \dots \Pi a_n. \sigma(t)$$

where it is implicitly assumed by the bound variable convention that no  $a_i$  is in the support of  $\sigma$  and none has a free occurrence in  $\sigma(b)$  for any  $b$  in the support of  $\sigma$ . We write  $\text{Ftv}(T)$  for the free type variables of a scheme  $T$ . The language includes one new construct for terms called a *let*. In an expression  $L \equiv \text{let } x = M \text{ in } N$ , free occurrences of  $x$  in  $M$  are free in  $L$  while those that occur in  $N$  are bound by the *let*.

The typing rules for  $\text{ML}_0$  will include a generalization of the projection rule in the implicit system that allows the type of a variable to be any type obtained by instantiating the  $\Pi$ -bound variables of a scheme associated with it in a type assignment.

**Definition 5.2.1.** A type  $s$  is said to be an *instance* of a type scheme  $T \equiv \Pi a_1 \dots \Pi a_n. t$  if there is a substitution  $\sigma$  with its support contained in  $\{a_1, \dots, a_n\}$  such that  $\sigma(t) = s$ . If  $s$  is an instance of  $T$  then we write  $s \leq T$ .

Assignments in  $\text{ML}_0$  are defined similarly to assignments for simple types, but an  $\text{ML}_0$  type assignment associates type *schemes* to term variables. Specifically, an *assignment* is a list  $H$  of pairs  $x : T$  where  $x$  is a term variable and  $T$  is a type scheme. The set of free type variables  $\text{Ftv}(H)$  in an assignment  $H$  is the union of the sets  $\text{Ftv}(H(x))$  where  $x \in H$ . To give the typing rules for the system it is necessary to define a notion of the *closure* of a type relative to an assignment. This is a function on assignments  $H$  and types  $t$  such that

$$\text{close}(H; t) = \Pi a_1 \dots \Pi a_n. t$$

where  $\{a_1, \dots, a_n\} = \text{Ftv}(t) - \text{Ftv}(H)$ . It is assumed that the function  $\text{close}$  chooses some particular order for the  $\Pi$  bindings here; it does not actually matter what this order is, but we can simply assume that our typing judgements are defined relative to a particular choice of the function  $\text{close}$ . A typing judgement is a triple  $H \Vdash M : t$  where  $H$  is an assignment,  $M$  a term, and  $t$  a type. The typing rules for the system appear in Table 13. The symbol  $\Vdash$  has been used in place of  $\vdash$  for this system to distinguish it from the implicit system of Table 12 and from another system to which it will be compared later (the one in Table 20 to be precise). The rules for abstraction and application are the same as for the implicit typing system. The rule [Proj] for variables is different though because the type of a variable  $x$  can be any instance of the type scheme  $H(x)$ . The rule [Let] for the *let* construct gives the type of the *let* as that of  $N$  in an assignment where the type associated with  $x$  is the closure of the type of  $M$ . Note that

**Table 13.** Typing rules for  $ML_0$ 


---

[Proj]	$\frac{x : T \in H \quad t \leq T}{H \Vdash x : t}$
[Abs] <sup>-</sup>	$\frac{H, x : s \Vdash M : t}{H \Vdash \lambda x. M : s \rightarrow t}$
[Appl]	$\frac{H \Vdash M : s \rightarrow t \quad H \Vdash N : s}{H \Vdash M(N) : t}$
[Let]	$\frac{H \Vdash M : s \quad H, x : \text{close}(H; s) \Vdash N : t}{H \Vdash \text{let } x = M \text{ in } N : t}$

---

there is a rule for each clause for a term in the grammar of the language, and the hypotheses of each rule are judgements about subterms of that term.

A basic property of the type variables and substitution in the system is given by the following:

**Lemma 5.2.2.** *If  $H \Vdash M : t$ , then  $[s/a]H \Vdash M : [s/a]t$ .*

In particular, if  $a \notin \text{Ftv}(H)$ , then  $[s/a]H \equiv H$ , so  $H \Vdash M : t$  implies  $H \Vdash M : [s/a]t$ .

As in the implicit simply-typed system,  $ML_0$  does not have a type for the term  $\lambda f. f(f)$ . However, if  $f$  is let-bound to an appropriate value  $M$ , then the term

$$N \equiv \text{let } f = M \text{ in } f(f)$$

can have a type. To see this in a specific example, take  $M$  to be the identity combinator  $\lambda x. x$ . Let  $a$  be a type variable, let us show that  $N$  has type  $a \rightarrow a$ . First of all,

$$\frac{x : a \Vdash x : a}{\Vdash \lambda x. x : a \rightarrow a}$$

follows from [Proj] and [Abs]<sup>-</sup>. Now, by [Proj], we must also have the hypotheses of the following instance of [Abs]<sup>-</sup>:

$$\frac{f : \Pi a. a \rightarrow a \Vdash f : a \rightarrow a \quad f : \Pi a. a \rightarrow a \Vdash f : (a \rightarrow a) \rightarrow (a \rightarrow a)}{f : \Pi a. a \rightarrow a \Vdash f(f) : a \rightarrow a}$$

Note, in particular, that it is possible to instantiate the  $\Pi$ -bound variable  $a$  as the type  $a \rightarrow a$  in one hypothesis and simply as  $a$  in the other. From the derivations above, we now have both hypotheses of this instance of the [Let] rule:

$$\frac{\vdash \lambda x. x : a \rightarrow a \quad f : \Pi a. a \rightarrow a \vdash f(f) : a \rightarrow a}{\vdash \text{let } f = \lambda x. x \text{ in } f(f) : a \rightarrow a}$$

One of the most important characteristics of this typing system is the fact that we can determine whether a term has a type in a given assignment. Given an assignment  $H$  and a substitution  $\sigma$ , let  $\sigma(H)$  be the assignment that associates  $\sigma(H(x))$  to each  $x \in H$ . That is, if  $H \equiv x_1 : T_1, \dots, x_n : T_n$ , then

$$\sigma(H) \equiv x_1 : \sigma(T_1), \dots, x_n : \sigma(T_n).$$

Given assignment  $H$  and term  $M$ , define  $\mathcal{S}(H, M)$  to be the set of all pairs  $(\sigma, t)$  such that  $\sigma$  is a substitution,  $t$  is a type, and  $\sigma(H) \vdash M : t$ . There is an algorithm that, given  $H$  and  $M$ , provides an element of  $\mathcal{S}(H, M)$  if the algorithm succeeds. To describe the algorithm, some background on substitutions is required.

Let  $\sigma$  and  $\tau$  be substitutions. Then  $\sigma$  is said to be *more general* than  $\tau$  if there is a substitution  $\sigma'$  such that  $\sigma' \circ \sigma = \tau$ . Given types  $s$  and  $t$ , a *unifier* for  $s, t$  is a substitution  $\sigma$  such that  $\sigma(s) = \sigma(t)$ . A *most general unifier* for  $s, t$  is a unifier  $\sigma$  that is more general than any other unifier for these types.

**Theorem 5.2.3.** *If there is a unifier for a pair of types, then there is also a most general unifier for them.*

This theorem and an algorithm for calculating most general unifiers was introduced in [Robinson, 1965]. The reason for introducing unifiers at this point is to describe an algorithm of Robin Milner [Milner, 1978] called *algorithm W*. It is given by induction on the structure of  $M$  by the following cases:

- Case  $M \equiv x$ . If  $x \in H$ , then the value is the identity substitution paired with the instantiation of the scheme  $H(x)$  by a collection of fresh type variables. In other words, if  $H(x) \equiv \Pi a_1 \dots \Pi a_n. s$  where  $a_1, \dots, a_n$  are new type variables, then

$$\mathcal{W}(H; x) = (\text{id}, s)$$

where  $\text{id}$  is the identity map. If  $x \notin H$ , then the value of  $\mathcal{W}(H; M)$  is failure.

- Case  $M \equiv \lambda x. M'$ . Suppose  $a$  is a new type variable and  $(\sigma, t) = \mathcal{W}(H, x : a; M')$ . Then

$$\mathcal{W}(H; \lambda x. M') = (\sigma, \sigma(a) \rightarrow t).$$

If, on the other hand, the value of  $\mathcal{W}(H, x : a; M')$  is failure, then so is  $\mathcal{W}(H; M)$ .



- Case  $M \equiv L(N)$ . Suppose  $(\sigma_1, t_1) = \mathcal{W}(H; L)$  and  $(\sigma_2, t_2) = \mathcal{W}(\sigma_1(H); N)$ . Let  $a$  be a new type variable. If there is a most general unifier  $\sigma$  for  $\sigma_2(t_1)$  and  $t_2 \rightarrow a$ , then

$$\mathcal{W}(H; L(N)) = (\sigma \circ \sigma_2 \circ \sigma_1, \sigma a).$$

In the event that there is no such unifier or if the value of  $\mathcal{W}(H; L)$  or  $\mathcal{W}(\sigma_1(H); N)$  is failure, then this is also the value of  $\mathcal{W}(H; M)$ .

- Case  $M \equiv \text{let } x = L \text{ in } N$ . Suppose  $(\sigma_1, s_1) = \mathcal{W}(H; L)$  and  $H' \equiv \sigma_1(H)$ ,  $x : \text{close}(\sigma_1(H); s_1)$ . If  $(\sigma_2, s_2) = \mathcal{W}(H'; N)$ , then

$$\mathcal{W}(H; M) = (\sigma_2 \circ \sigma_1, s_2).$$

If, on the other hand, the value of  $\mathcal{W}(H; L)$  or  $\mathcal{W}(H'; N)$  is failure, then that is also the value of  $\mathcal{W}(H; M)$ .

To prove that algorithm  $\mathcal{W}$  is sound, it helps to have the following:

**Lemma 5.2.4.** *If  $H, x : \Pi a_1 \dots \Pi a_n. s \Vdash M : t$ , then  $H, \text{close}(H; s) \Vdash M : t$ .*

To understand this, note that the bound variable convention insists that the variables  $a_i$  that appear in  $s$  are not in  $\text{Ftv}(H)$ . The desired soundness property can be stated precisely as follows:

**Theorem 5.2.5.** *If  $\mathcal{W}(H; M)$  exists, then it is an element of  $\mathcal{S}(H; M)$ .*

**Proof.** Suppose  $\mathcal{W}(H; M)$  exists; we must show that  $\sigma(H) \Vdash M : t$ . The proof is by induction on the structure of  $M$ . If  $M \equiv x$  is a variable, then  $t$  is an instantiation of  $H(x)$ . This means  $H \Vdash x : t$  by the typing rule for variables.

Case  $M \equiv \lambda x. M'$ . If  $(\sigma, t) = \mathcal{W}(H, x : a; M')$ , then  $\sigma(H, x : a) \Vdash M' : t$  by the inductive hypothesis. Thus  $\sigma(H), x : \sigma(a) \Vdash M' : t$  so  $\sigma(H) \Vdash \lambda x. M' : \sigma(a) \rightarrow t$  by the typing rule for abstraction. This means  $\mathcal{W}(H; M) = (\sigma(H), \sigma(a) \rightarrow t) \in \mathcal{S}(H; M)$ .

Case  $M \equiv L(N)$ . If  $(\sigma_1, t_1) = \mathcal{W}(H; L)$  and  $(\sigma_2, t_2) = \mathcal{W}(\sigma_1(H); N)$ , then, by the inductive hypothesis,  $\sigma_1(H) \Vdash L : t_1$  and  $\sigma_2(\sigma_1(H)) \Vdash N : t_2$ . Since  $\mathcal{W}(H; M)$  exists, there is a substitution  $\sigma$  such that  $\sigma(\sigma_2(t_1)) \equiv \sigma(t_2 \rightarrow a) \equiv \sigma(t_2) \rightarrow \sigma(a)$ . By Lemma 5.2.2,  $\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash L : \sigma \circ \sigma_2(t_1)$  so

$$\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash L : \sigma(t_2) \rightarrow \sigma(a).$$

Also by the inductive hypothesis,

$$\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash N : \sigma(t_2).$$

Combining these facts with the rule for the typing of applications, we can conclude that

$$\sigma \circ \sigma_2 \circ \sigma_1(H) \Vdash L(N) : \sigma(a),$$

which means that  $\mathcal{W}(H; M) \in \mathcal{S}(H; M)$ .

Case  $M \equiv \text{let } x = L \text{ in } N$ . If  $(\sigma_1, s_1) = \mathcal{W}(H; L)$  and  $(\sigma_2, s_2) = \mathcal{W}(H'; N)$  where  $H' = \sigma_1(H), x : \text{close}(\sigma_1(H), s_1)$ , then, by the inductive hypothesis,  $\sigma_1(H) \Vdash L : s_1$  and  $\sigma_2(H') \Vdash N : s_2$ . By Lemma 5.2.2,

$$\sigma_2 \circ \sigma_1(H) \Vdash L : \sigma_2(s_1).$$

If  $\Pi a_1 \dots \Pi a_n. s_1 \equiv \text{close}(\sigma_1(H), s_1)$ , then

$$\sigma_2 \circ \sigma_1(H), x : \Pi a_1 \dots \Pi a_n. \sigma_2(s_1) \Vdash N : s_2$$

so, by Lemma 5.2.4,

$$\sigma_2 \circ \sigma_1(H), x : \text{close}(\sigma_2 \circ \sigma_1(H), \sigma_2(s_1)) \Vdash N : s_2.$$

By the rule for typing lets, this says that  $\mathcal{W}(H; M) = (\sigma_2 \circ \sigma_1, s_2) \in \mathcal{S}(H; M)$ . ■

Of course, this only proves that the answer calculated by  $\mathcal{W}$  is ‘sound’; another question, addressed in [Damas and Milner, 1982] led to the formulation of a theorem describing the sense in which the type inferred by algorithm  $\mathcal{W}$  is the ‘best’ type possible. To be precise,

**Definition 5.2.6.** A *principal type* for  $H$  and  $M$  is a pair  $(\sigma, s) \in \mathcal{S}(H, M)$  such that, for any other pair  $(\tau, t) \in \mathcal{S}(H, M)$ , there is a substitution  $\sigma'$  such that

- $\tau(H) = \sigma' \circ \sigma(H)$  and
- $t$  is an instance of  $\sigma'(\text{close}(\sigma(H); s))$ .

In the case that  $H$  is the empty assignment, this boils down to saying that if a closed term  $M$  has a type at all, then there is a type  $s$  such that  $M$  has type  $s$  and, for any other type  $t$ ,  $M$  has type  $t$  only if  $t = \tau s$  for some substitution  $\tau$ . Damas and Milner state the following:

**Theorem 5.2.7.** *If there is a type  $t$  such that  $H \Vdash M : t$ , then  $\mathcal{W}(H; M)$  is a principal type scheme for  $H$  and  $M$ .*

Its proof appears in the thesis [Damas, 1985]. In practice there are many optimizations that can be done to provide a more efficient implementation of principal type scheme inference. One discussion of implementation has appears in [Cardelli, 1987]. Most books on ML include some discussion of ML type inference; for instance [Sokolowski, 1991] provides code for unification and inference using the Standard ML module system.

### 5.3 Run-time safety for assignments and continuations.

To appreciate the value of Algorithm  $\mathcal{W}$ , it is necessary to consider it in connection with the way in which programs will be evaluated. This was

not specified above, but this does not mean that the matter is trivial or unimportant; on the contrary, this topic requires some careful consideration if pitfalls are to be avoided. A straightforward approach to the semantics of a term **let**  $x = N$  **in**  $M$  is to treat it *operationally* as an application  $(\lambda x. M)N$ . The typing system of  $ML_0$  treats these two terms differently, but it is possible to treat them as the same operationally. This is the approach taken by the Standard ML definition. It leads to complexities, however, when one considers the interaction between certain computational extensions and evaluation order. Let us now consider this issue and look at one possible solution proposed in [Leroy, 1993].

Let us assume that our language is to be given a call-by-value evaluation order and extended with the primitives and the rule for pairs that appear in Table 14. If desired, a sequencing operator can be included by taking

**Table 14.** Assignments, continuations, and pairs

---

$\text{ref} : \Pi a. a \rightarrow \text{ref}(a)$   
 $\text{deref} : \Pi a. \text{ref}(a) \rightarrow a$   
 $\text{update} : \Pi a. (\text{ref}(a) \times a) \rightarrow a$   
 $\text{callcc} : \Pi a. (\text{cont}(a) \rightarrow a) \rightarrow a$   
 $\text{throw} : \Pi a. \Pi b. (a \times \text{cont}(a)) \rightarrow b$   
 $\text{fst} : \Pi a. \Pi b. (a \times b) \rightarrow a$   
 $\text{snd} : \Pi a. \Pi b. (a \times b) \rightarrow b$

$$\frac{H \vdash M : s \quad H \vdash N : t}{H \vdash (M, N) : s \times t}$$


---

$M; N$  to be syntactic sugar for  $(\lambda x. N)M$  where  $x$  does not appear in  $N$ .

Algorithm  $\mathcal{W}$  can easily be adapted to deal with these extra constructs. The new primitives can be treated as if they were free variables with the appropriate type schemes, and the inference for pairs can be done component-wise. Unfortunately this approach causes difficulties when taken with the usual evaluation order for the **let** construct, if this construct is viewed as syntactic sugar for an application as described above. A classic example of the difficulty with the use of Algorithm  $\mathcal{W}$  for this language is given by the following program, which is written in a pseudo-code compromise between Standard ML and  $ML_0$ :

```

let r = ref(fn x => x)
in update(r, fn x => x+1);

```

```
(deref r)(true)
end
```

This program will pass Algorithm  $\mathcal{W}$  because the type of  $r$  will be

$$\Pi a. \text{ref}(a \rightarrow a).$$

Suppose the three primitives **ref**, **update**, and **deref** are treated as having the semantics of their Standard ML analogs **ref**, **:=**, and **!**, and the semantics of **let** is the one of SML. Then there will be a run-time error when  $r$  is dereferenced and an attempt is made to add 1 to **true**. The problem here is that the **update** operation on  $r$  instantiates the type of  $r$  so that the type of the dereferenced value is an inappropriate specialization of the polymorphic type of  $r$  at the point it is applied to **true**.

A similar problem arises with the addition of the control primitives **callcc** and **throw** to the language. Let us suppose that these primitives are given the semantics that **callcc** and the application of continuations have in Scheme. Duba, Harper, and MacQueen [Duba *et al.*, 1991] noted that Algorithm  $\mathcal{W}$  may typecheck programs with these constructs that yield run-time errors. Here is an example from [Leroy, 1993]:

```
let later = callcc(fn k =>
                    (fn x => x,
                     fn f => throw(k, (f, fn g => 1))))
in print(first(later)("Hello World"));
   second(later)(fn x => x+1)
end
```

When invoked, the continuation  $k$  essentially carries out a reassignment of the identifier **later**. To see this and why it is a problem, let us trace the evaluation. After the **let** binding of **later**, the **first** coordinate of **later**—the identity function—will be invoked on a string, which is printed. Then the **second** coordinate is invoked on a function which is ‘thrown’ as the first coordinate of a pair to the previously captured continuation  $k$ . At this point, the evaluation proceeds in the same manner as the following program:

```
let later = (fn x => x+1, fn g => 1)
in print(first(later)("Hello World"));
   second(later)(fn x => x+1)
end
```

which leads to a type error when **first(later)** is applied because this leads to the addition of 1 and a string.

There have been a variety of proposals about how to deal with these problems by restricting the ML type system [Damas, 1985; Tofte, 1990]. An alternative is to change the evaluation order of the **let** expressions themselves. To illustrate this idea, let us consider an extension of  $\text{ML}_0$

which we call  $ML_1$ . It has the following grammar:

$$\begin{aligned}
 x &\in \text{TermVariable} \\
 a &\in \text{TypeVariable} \\
 t &::= a \mid t \rightarrow t \mid t \times t \mid \mathbf{ref}(t) \mid \mathbf{cont}(t) \\
 T &::= t \mid \Pi a. T \\
 M &::= x \mid \lambda x. M \mid MM \mid \mathbf{let } x = M \mathbf{ in } M \mid (M, M) \mid C \\
 C &::= \mathbf{ref} \mid \mathbf{deref} \mid \mathbf{update} \mid \mathbf{callcc} \mid \mathbf{throw} \mid \mathbf{fst} \mid \mathbf{snd}
 \end{aligned}$$

and its typing rules are those of  $ML_0$  together with the types for the constants and pairs as given above.

The inclusion of assignments in the language leads us to incorporate notions of environment and store into our semantics, while the inclusion of first class continuations leads us to consider control as well. In earlier semantics the idea of a store could be ignored. Environments were avoided by using syntactic substitution to instantiate formal parameters to actual parameters, and control was expressed indirectly through the hypotheses of rules in SOS or natural semantics. Let us now consider an alternative way of describing a semantics which is essentially an *abstract machine*. The machine is described through a collection of transition rules in which environments, store, and control are all made explicit. The machine here is essentially a reformulation of the natural semantics appearing in [Leroy, 1993]. It closely resembles the abstract machine in [Felleisen and Friedman, 1986].

To describe it, we require some further concepts. The following grammar defines values  $V$ , continuations  $\kappa$ , and thunks  $T$  in terms of environments  $\rho$  and stores  $\sigma$ :

$$\begin{aligned}
 l &\in \text{Location} \\
 V &::= \mathbf{Closure}(M, \rho) \mid (V, V) \mid l \mid \kappa \mid C \\
 \kappa &::= \mathbf{Stop} \mid \mathbf{Apply1}(M, \rho, \kappa) \mid \mathbf{Apply2}(V, \kappa) \mid \\
 &\quad \mathbf{Pair1}(M, \rho, \kappa) \mid \mathbf{Pair2}(V, \kappa) \\
 T &::= \mathbf{Thunk}(M, \rho)
 \end{aligned}$$

An *environment*  $\rho$  is partial function with finite domain called an *environment* that maps a variable  $x$  to a value  $V$  or a thunk  $T$ . A store  $\sigma$  is a partial function with finite domain called a *store* that maps locations  $l$  to values  $V$ .

The machine is given in terms of a set of rewrite rules involving two operators push and pop. The operator  $\text{push}(M, \rho, \sigma, \kappa)$  takes a term  $M$ , an environment  $\rho$ , a store  $\sigma$ , and a continuation  $\kappa$  as its arguments. The operator  $\text{pop}(V, \sigma, \kappa)$  takes a value  $V$ , a store  $\sigma$ , and a continuation  $\kappa$  as arguments. If they terminate, the rewrite rules produce a pair  $(V, \sigma)$  or type error, **tyerr**, at the end. The relation  $\rightarrow$  is defined to be the least one satisfying the rules in Table 15. We assume we are given a function



Table 15. Abstract Machine for  $ML_1$ 


---

$\text{push}(x, \rho, \sigma, \kappa) \rightarrow \text{pop}(\rho(x), \sigma, \kappa)$	if $\rho(x)$ is a value
$\text{push}(x, \rho, \sigma, \kappa) \rightarrow \text{push}(M, \rho', \sigma, \kappa)$	if $\rho(x) = \mathbf{Thunk}(M, \rho')$
$\text{push}(\lambda x. M, \rho, \sigma, \kappa) \rightarrow \text{pop}(\mathbf{Closure}(\lambda x. M, \rho), \sigma, \kappa)$	
$\text{push}(MN, \rho, \sigma, \kappa) \rightarrow \text{push}(M, \rho, \sigma, \mathbf{Apply1}(N, \rho, \kappa))$	
$\text{push}(C, \rho, \sigma, \kappa) \rightarrow \text{pop}(C, \sigma, \kappa)$	
$\text{push}((M, N), \rho, \sigma, \kappa) \rightarrow \text{push}(M, \rho, \sigma, \mathbf{Pair1}(N, \rho, \kappa))$	
$\text{push}(\mathbf{let } x = M \mathbf{ in } N, \rho, \sigma, \kappa) \rightarrow \text{push}(N, \rho[x \mapsto \mathbf{Thunk}(M, \rho)], \sigma, \kappa)$	
$\text{push}(M, \rho, \sigma, \kappa) \rightarrow \mathbf{tyerr}$	in all other cases
<hr/>	
$\text{pop}(V, \sigma, \mathbf{Apply1}(N, \rho, \kappa)) \rightarrow \text{push}(N, \rho, \sigma, \mathbf{Apply2}(V, \kappa))$	
$\text{pop}(V, \sigma, \mathbf{Apply2}(\mathbf{Closure}(\lambda x. M, \rho), \kappa)) \rightarrow \text{push}(M, \rho[x \mapsto V], \sigma, \kappa)$	
$\text{pop}(V, \sigma, \mathbf{Apply2}(\mathbf{ref}, \kappa)) \rightarrow \text{pop}(\mathbf{new}(\sigma), \sigma[\mathbf{new}(\sigma) \mapsto V], \kappa)$	
$\text{pop}(l, \sigma, \mathbf{Apply2}(\mathbf{deref}, \kappa)) \rightarrow \text{pop}(\sigma(l), \sigma, \kappa)$	
$\text{pop}((l, V), \sigma, \mathbf{Apply2}(\mathbf{update}, \kappa)) \rightarrow \text{pop}(V, \sigma[l \mapsto V], \kappa)$	
$\text{pop}(\mathbf{Closure}(\lambda k. M, \rho), \sigma, \mathbf{Apply2}(\mathbf{callcc}, \kappa)) \rightarrow$ $\text{push}(M, \rho[k \mapsto \kappa], \sigma, \kappa)$	
$\text{pop}((V, \kappa'), \sigma, \mathbf{Apply2}(\mathbf{throw}, \kappa)) \rightarrow \text{pop}(V, \sigma, \kappa')$	
$\text{pop}((U, V), \sigma, \mathbf{Apply2}(\mathbf{fst}, \kappa)) \rightarrow \text{pop}(U, \sigma, \kappa)$	
$\text{pop}((U, V), \sigma, \mathbf{Apply2}(\mathbf{snd}, \kappa)) \rightarrow \text{pop}(V, \sigma, \kappa)$	
$\text{pop}(U, \sigma, \mathbf{Pair1}(N, \rho, \kappa)) \rightarrow \text{push}(N, \rho, \sigma, \mathbf{Pair2}(U, \kappa))$	
$\text{pop}(V, \sigma, \mathbf{Pair2}(U, \kappa)) \rightarrow \text{pop}((U, V), \sigma, \kappa)$	
$\text{pop}(V, \sigma, \mathbf{Stop}) \rightarrow (V, \sigma)$	
$\text{pop}(V, \sigma, \kappa) \rightarrow \mathbf{tyerr}$	in all other cases

---

**new** from stores to locations such that  $\mathbf{new}(\sigma)$  is a location not in the domain of definition of  $\sigma$ . This function is used in the semantics of the **ref** constant for allocating new locations in memory. Intuitively, the push machine evaluates a term in stages, delaying parts of the calculation by pushing chores onto  $\kappa$ , which represents the continuation stack. When this evaluation reaches a value, the pop machine is invoked to consult the continuation stack to determine what should be done with the value.

The key result is the following:

**Theorem 5.3.1.** *If  $M$  is type correct, then it is not the case that*

$$\text{push}(M, \emptyset, \emptyset, \mathbf{Stop}) \rightarrow^* \mathbf{tyerr}.$$

The proof requires establishing a set of type invariants for environments, stores, and continuations. A further fact that can be shown is that, given a suitable formulation of types for values, if  $\text{push}(M, \emptyset, \emptyset, \text{Stop}) \rightarrow^* (V, \sigma)$ , then the type of  $V$  in  $\sigma$  is the same as that of  $M$ .

Another approach to dealing with the semantics of **let** is to restrict the declaration so that polymorphism is only permitted in expressions that do not require any evaluation. This would make it illegal to write programs such as

**let**  $f = g \circ h$  **in** ...

where  $\circ$  is the higher-order composition function, but one could use an  $\eta$ -expansion and instead write

**let**  $f = \text{fn } x \Rightarrow g(h\ x)$  **in** ...

Andrew Wright, who introduced this idea [Wright, 1993], has been able to demonstrate its practicality for a number of substantial SML programs.

## 6 Types as subsets

So far we have modeled types as objects drawn from a collection of spaces or as syntactic invariants. Each closed, well-typed term denotes a value in the space that interprets its type or expresses a property that is unchanged by the evaluation of a term. Let us now consider another perspective in which a type is viewed as a subset of a *universe* of elements modeling terms. In this approach, the meaning  $\llbracket M \rrbracket$  of a closed, well-typed term  $M$  is a member of the universe  $U$ , and  $\llbracket M \rrbracket$  lies in the subset  $\llbracket t \rrbracket \subseteq U$  of the universe that interprets the type  $t$  of  $M$ . There are several ways to view these subsets, resulting in different models. Our discussion begins with an examination of the interpretation of the *untyped*  $\lambda$ -calculus using a model of the simply-typed calculus that satisfies a special equation. We then consider how a model of the untyped calculus can be viewed as a model of the typed one by interpreting types as subsets.

### 6.1 Untyped $\lambda$ -calculus

The *untyped*  $\lambda$ -calculus is essentially the calculus obtained by removing the type system from the simply-typed calculus. The terms of the untyped calculus are generated by the following grammar:

$$M ::= x \mid MM \mid \lambda x. M$$

where the abstraction of the variable  $x$  over a term  $M$  binds the free variable occurrences of  $x$  in  $M$ . (These are the same terms used for the implicitly-typed system before.) Expressions such as  $M', N, N_1, \dots$  are used to range over untyped  $\lambda$ -terms as well as typed terms—context must determine which class of terms is intended. For discussions below relating

typed and untyped calculi,  $P, Q, R$  are used for terms with type tags and  $L, M, N$  for those without such tags. Untyped  $\lambda$ -terms are subject to the same conventions about bound variables as we applied earlier to terms with type tags on bound variables. In particular, terms are considered equivalent if they are the same up to the renaming of bound variables (where no such renaming leads to capture of a variable by a new binding). The equational rules for the untyped  $\lambda\beta$ -calculus are given in Table 16. They

**Table 16.** Equational rules for Untyped  $\lambda\beta$ -calculus

$\{\text{Refl}\}$	$x = x$
$\{\text{Sym}\}$	$\frac{M = N}{N = M}$
$\{\text{Trans}\}$	$\frac{L = M \quad M = N}{L = N}$
$\{\text{Cong}\}$	$\frac{M = M' \quad N = N'}{H \vdash M(N) = M'(N')}$
$\{\xi\}$	$\frac{M = M'}{\lambda x. M = \lambda x. M'}$
$\{\beta\}$	$(\lambda x : s. M)(N) = [N/x]M$

are very similar to those of the typed calculus. The untyped  $\lambda\beta\eta$ -calculus is obtained by including the untyped version of the  $\eta$ -rule,

$$\{\eta\} \quad \lambda x. M(x) = M,$$

where, as before, the variable  $x$  does not appear free in the expression  $M$ . It is not hard to see that every term of the simply-typed  $\lambda$ -calculus gives rise to a term of the untyped calculus obtained by ‘erasing’ the type tags on its free variables. More precisely, we define the *erasure*  $\text{erase}(P)$  of a term  $P$  of the simply-typed calculus by induction as follows:

$$\begin{aligned} \text{erase}(x) &\equiv x \\ \text{erase}(P(Q)) &\equiv (\text{erase}(P))(\text{erase}(Q)) \\ \text{erase}(\lambda x : t. P) &\equiv \lambda x. \text{erase}(P). \end{aligned}$$

Although every term of the untyped calculus can be obtained as the erasure of a tagged one, it is not the case that every untyped term can be obtained as the erasure of a *well-typed* tagged term. For example, if  $\text{erase}(P) \equiv$

$\lambda x. x(x)$ , then there is no context  $H$  and type expression  $t$  such that  $H \vdash P : t$ . Of course, distinct well-typed terms may have the same erasure if they differ only in the tags on their bound variables:

$$\text{erase}(\lambda x : \mathbf{o}. x) \equiv \lambda x. x \equiv \text{erase}(\lambda x : \mathbf{o} \rightarrow \mathbf{o}. x).$$

## 6.2 What is a model of the untyped $\lambda$ -calculus?

It is possible to describe a semantics for the untyped  $\lambda$ -calculus using the simply-typed calculus. Such an interpretation must deal with the concept of self application such as we see in the term  $\lambda x. x(x)$ , so some care must be applied in explaining how an untyped term can be interpreted in a typed setting where this operation is not type-correct. The approach we use, which follows [Meyer, 1982], is to view the application as entailing an *implicit coercion* that converts an application instance of a value into a corresponding function. More precisely, assume we are given constants

$$\begin{aligned}\Phi &: \mathbf{o} \rightarrow (\mathbf{o} \rightarrow \mathbf{o}) \\ \Psi &: (\mathbf{o} \rightarrow \mathbf{o}) \rightarrow \mathbf{o}\end{aligned}$$

and an equational theory with one equation

$$\Phi \circ \Psi = \lambda x : \mathbf{o} \rightarrow \mathbf{o}. x. \quad (6.1)$$

Let us call this *theory*  $U^\beta$ . A model of theory  $U^\beta$  is a tuple

$$(\mathcal{A}, A, \Phi, \Psi)$$

where  $(\mathcal{A}, A)$  is a type frame and  $\Phi \in D^{\mathbf{o} \multimap (\mathbf{o} \multimap \mathbf{o})}$  and  $\Psi \in D^{(\mathbf{o} \multimap \mathbf{o}) \multimap \mathbf{o}}$  satisfy (6.1). (To simplify the notation, let us make no distinction between  $\Phi$  and  $\Psi$  as constant symbols in the calculus and their interpretations in the model.) We may use a model of theory  $U^\beta$  to interpret the untyped  $\lambda\beta$ -calculus in the following way. First, we define a *syntactic translation* that converts an untyped term into a term with type tags by induction as follows:

$$\begin{aligned}x^* &\equiv x \\ (\lambda x. M)^* &\equiv \Psi(\lambda x : \mathbf{o}. M^*) \\ (M(N))^* &\equiv \Phi(M^*)(N^*)\end{aligned}$$

For example,  $Y$  and  $Y^*$  are as follows:

$$\begin{aligned}Y &\equiv \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \\ Y^* &\equiv \\ &\Psi(\lambda f : \mathbf{o}. \Phi(\Psi(\lambda x : \mathbf{o}. \Phi(f)(\Phi(x)(x))))(\Psi(\lambda x : \mathbf{o}. \Phi(f)(\Phi(x)(x)))).\end{aligned}$$

It is possible to demonstrate the following basic fact about the translation:

**Proposition 6.2.1.** *Let  $M$  be an untyped term. If  $H \equiv x_1 : \mathbf{o}, \dots, x_n : \mathbf{o}$  is a type context that includes all of the free variables of  $M$ , then  $H \vdash M^* : \mathbf{o}$ .*

With this translation, it is possible to assign a meaning to an untyped term  $M$  by taking  $\mathcal{A}_u[M] = \mathcal{A}[M^*] \in D^{\mathbf{o}}$ . To see that this respects the equational rules given in Table 16, note first the following:

**Lemma 6.2.2.**  $[N^*/x]M^* \equiv ([N/x]M)^*$ .

We then prove the  $\beta$ -rule for the untyped calculus by a calculation in the typed one:

$$\begin{aligned} ((\lambda x. M)(N))^* &= \Phi((\lambda x. M)^*)(N^*) \\ &\equiv \Phi(\Psi(\lambda x : \mathbf{o}. M^*))(N^*) \\ &\equiv (\lambda x : \mathbf{o}. M^*)(N^*) \\ &= [N^*/x]M^* \\ &\equiv ([N/x]M)^* \end{aligned}$$

so

$$\mathcal{A}_u[(\lambda x. M)(N)] = \mathcal{A}[(\lambda x. M)(N))^*] = \mathcal{A}[(N/x]M)^*] = \mathcal{A}_u[[N/x]M].$$

The other axioms and rules are not difficult.

Now, let *theory*  $U^{\beta\eta}$  be theory  $U^\beta$  together with the equation

$$\Psi \circ \Phi = \lambda f : \mathbf{o}. f, \tag{6.2}$$

which asserts, in effect, that  $\Phi$  is an isomorphism between the ground type  $D^{\mathbf{o}}$  and the functions in  $D^{\mathbf{o} \rightarrow \mathbf{o}}$ . If a model  $\mathcal{A}$  of theory  $U^\beta$  is also a model of theory  $U^{\beta\eta}$ , then  $\mathcal{A}_u$  satisfies the  $\eta$ -rule as well as the  $\beta$ -rule. To see this, suppose the variable  $x$  does not appear free in the term  $M$ , then

$$\begin{aligned} (\lambda x. M(x))^* &\equiv \Psi(\lambda x : \mathbf{o}. (M(x))^*) \\ &\equiv \Psi(\lambda x : \mathbf{o}. \Phi(M^*)(x)) \\ &= \Psi(\Phi(M^*)) \\ &= M^* \end{aligned}$$

so  $\mathcal{A}_u[\lambda x. M(x)] = \mathcal{A}[(\lambda x. M(x))^*] = \mathcal{A}[M^*] = \mathcal{A}_u[M]$ . In summary, we have the following result.

**Theorem 6.2.3.** *If  $\mathcal{A}$  is a model of  $U^\beta$ , then  $\mathcal{A}_u$  is a model of the untyped  $\lambda\beta$ -calculus. If, moreover,  $\mathcal{A}$  is a model of  $U^{\beta\eta}$ , then it is a model of the untyped  $\lambda\beta\eta$ -calculus.*

### 6.3 What models of the untyped $\lambda$ -calculus are there?

Having established that models of theories of  $U^\beta$  and  $U^{\beta\eta}$  provide models of the untyped  $\lambda\beta$  and  $\lambda\beta\eta$  calculi respectively, it is tempting to conclude that



we have almost completed our quest for models of the untyped calculus. In fact, we have only done the *easy* part. We have not yet shown that any models of theories  $U^\beta$  and  $U^{\beta\eta}$  actually *exist*. To see why this might be a problem, consider the full type frame  $\mathcal{F}_X$  over a set  $X$ . If we can find functions

$$\begin{aligned}\Phi &: X \rightarrow (X \rightarrow X) \\ \Psi &: (X \rightarrow X) \rightarrow X\end{aligned}$$

(where  $X \rightarrow X$  is the set of all functions from  $X$  to  $X$ ) that satisfy Equation (6.1), then we have produced the desired model. But Equation (6.1) implies that the function  $\Phi$  is a *surjection* from  $X$  onto the set of functions  $f : X \rightarrow X$ . By Cantor's theorem, such a surjection exists only if  $X$  has exactly one point! This means that the full type frame can only yield a trivial model for the untyped calculus through a choice of  $\Phi$  and  $\Psi$ . The problem lies in the fact that the interpretation of  $\mathbf{o} \rightarrow \mathbf{o}$  in the full type frame has *too many functions*. To find a model of the untyped calculus, we must therefore look for a type frame that has a more parsimonious interpretation of the higher types.

Techniques for finding models that can satisfy these properties were the primary purpose of the theory of domains and it is beyond the scope of this chapter to discuss how this can be done in any detail. For the sake of concreteness, however, let us build one such domain explicitly. Given a set  $X$ , let  $\mathcal{P}_f(X)$  be the set of all finite subsets of  $X$ . Define an operation  $G$  by

$$G(X) = \{(u, x) \mid u \in \mathcal{P}_f(X) \text{ and } x \in X\}.$$

Starting with any set  $X$ , let  $X_0 = X$  and  $X_{n+1} = G(X_n)$ . Take  $D_X$  to be the set  $\mathcal{P}(\bigcup_{n \in \omega} X_n)$  of all subsets of the union of the  $X_n$ s. Ordered by set inclusion, this is an algebraic lattice whose compact elements are the finite sets. To see how it can be viewed as a model of the untyped  $\lambda$ -calculus, consider an element  $(u, x) \in G(X_n)$ . This pair can be viewed as a piece of a function  $f$ , which has  $x$  in its output whenever  $u$  is a subset of its input. Suppose that  $d \in D_X$ . Following this intuition, we define a function  $\Phi(d) : D_X \rightarrow D_X$  by taking

$$\Phi(d)(e) = \{x \mid u \subseteq e \text{ for some } (u, x) \in d\} \quad (6.3)$$

for each  $e \in D_X$ . In other words, if we are to view  $d$  as inducing a function taking  $e$  as its argument, the result of applying  $d$  to  $e$  is the set of those elements  $x$  such that there is a 'piece' (element)  $(u, x)$  of  $d$  where  $u$  is a subset of the input  $e$ . Also, given a continuous function  $f : D_X \rightarrow D_X$ , define  $\Psi(f) \in D_X$  by

$$\Psi(f) = \{(u, x) \mid x \in f(u)\}. \quad (6.4)$$

This says that  $f$  is to be represented by recording the pairs  $(u, x)$  such that  $x$  will be part of the result of applying  $f$  to an element that contains  $u$ . It is not difficult to check that  $\Phi$  and  $\Psi$  are continuous. Suppose that  $f : D_X \rightarrow D_X$  is continuous. Then

$$\begin{aligned} \Phi(\Psi(f))(d) &= \{x \mid (u, x) \in \Psi(f) \text{ for some } u \subseteq d\} \\ &= \{x \mid x \in f(u) \text{ for some } u \subseteq d\} \\ &= \bigcup \{f(u) \mid u \subseteq d\} \\ &= f(d) \end{aligned}$$

where the last step follows from the fact that a continuous function on an algebraic cpo is determined by its action on compact elements.

A model of the simply-typed  $\lambda$ -calculus is defined by taking  $D_X$  as the interpretation of the ground type and interpreting higher types using the (pointwise ordered) continuous function spaces. It then follows from the calculation above that the continuous type frame generated by  $D_X$ , together with the continuous functions  $\Phi$  and  $\Psi$ , is a model of the untyped  $\lambda\beta$ -calculus. It is obviously non-trivial if  $X$  has at least two distinct elements.

Could  $D_X$  also be a model of the untyped  $\lambda\beta\eta$ -calculus? Suppose  $d \in D_X$ , and let us attempt to calculate equation (6.2):

$$\begin{aligned} \Psi(\Phi(d)) &= \{(u, x) \mid x \in \Phi(d)(u)\} \\ &= \{(u, x) \mid v \subseteq u \text{ for some } (v, x) \in d\} \\ &\supseteq d. \end{aligned}$$

But since  $d$  could be an arbitrary subset of  $\bigcup_n X_n$ , it is clear that this superset relation may fail to be an *equality*. So we have not yet demonstrated a model for the  $\lambda\beta\eta$ -calculus! To do this using cpos and continuous functions, what we need is a cpo  $D$  such that  $D \cong [D \rightarrow D]$ . Here is where domain theory would be helpful in finding spaces with the needed properties. Let us assume that we know how to find non-trivial domains to satisfy this and other isomorphisms; for details on how such domains can be constructed, see [Abramsky and Jung, 1994].

## 6.4 Inclusive subsets as types

A domain used as a universe for interpreting types is generally called a *universal domain*. It is not desirable to use arbitrary subsets of a universal domain to denote types because certain properties are needed to establish invariants. Let us now consider one of the most widely used conditions.

**Definition 6.4.1.** A subset of a cpo is *inclusive* if it is downward closed and closed under least upper bound's of  $\omega$ -chains.

The use of the term 'inclusive' for this notion is slightly non-standard from a historical perspective. The condition above was introduced in [Mil-

ner, 1978], and it is common for the term ‘ideal’ to be used instead [MacQueen *et al.*, 1986]. This clashes with another common usage of ‘ideal’, so the alternative term ‘inclusive’ is used here.

Based on a given universal domain, it is possible to use inclusive subsets to model simple types. Let  $X$  be any domain and suppose we are given a solution to the domain equation

$$D \cong X \oplus [D \rightarrow D]. \quad (6.5)$$

In this equation, the operator  $\oplus$  is the *coalesced sum*, which takes the disjoint union of its arguments and then identifies their respective least elements. This is a model of the untyped  $\lambda$ -calculus: let  $\Psi : [D \rightarrow D] \rightarrow D$  be the injection of the cpo of continuous functions from  $D$  to  $D$  into the right component of  $D$  (note that the function  $x \mapsto \perp_D$  is being set to  $\perp_D$ ) and let  $\Phi : D \rightarrow [D \rightarrow D]$  be given by

$$\Phi(y) = \begin{cases} f & \text{if } y = \Psi(f) \text{ for a continuous } f : D \rightarrow D \\ \perp_{[D \rightarrow D]} & \text{if } y \in X. \end{cases}$$

It is easy to see that  $\Phi \circ \Psi = \text{id}$ . To associate inclusive subsets on  $D$  with types of the simply-typed  $\lambda$ -calculus, define

$$\begin{aligned} \llbracket \mathbf{o} \rrbracket &= X \\ \llbracket s \rightarrow t \rrbracket &= \{ \Psi(f) \mid f \in [D \rightarrow D] \text{ and } f(x) \in \llbracket t \rrbracket \text{ whenever } x \in \llbracket s \rrbracket \}. \end{aligned}$$

**Lemma 6.4.2.** *For each type  $t$ , the subset  $\llbracket t \rrbracket \subseteq D$  is an inclusive subset.*

Given a type assignment  $H$ , an  $H$ -environment  $\rho$  is defined to be a function from variables into  $D$  such that  $\rho(x) \in \llbracket H(x) \rrbracket$  for each  $x \in H$ . If  $H \vdash M : t$ , then the interpretation of  $M$  is given by induction on the structure of  $M$  relative to an  $H$ -environment  $\rho$  as follows:

$$\begin{aligned} \llbracket x \rrbracket \rho &= \rho(x) \\ \llbracket L(N) \rrbracket \rho &= \Phi(\llbracket L \rrbracket \rho)(\llbracket N \rrbracket \rho) \\ \llbracket \lambda x : s. M' \rrbracket \rho &= \Psi(d \mapsto \llbracket M' \rrbracket \rho[x \mapsto d]) \end{aligned}$$

The key result relating this interpretation to the interpretations of types is the following:

**Theorem 6.4.3.** *If  $H \vdash M : t$  and  $\rho$  is an  $H$ -environment, then  $\llbracket M \rrbracket \rho \in \llbracket t \rrbracket$ .*

There is a problem with interpreting types in this way, however: the equational rules are not all satisfied. To see why this is the case, suppose that the domain  $X$  in Equation (6.5) has at least one element other than  $\perp_X$ . Define two terms

$$\begin{aligned} M &\equiv \lambda y : s \rightarrow t. \lambda x : s. y(x) \\ N &\equiv \lambda y : s \rightarrow t. y. \end{aligned}$$

The meaning of  $M$  can be calculated as

$$\begin{aligned} \llbracket M \rrbracket \emptyset &= \Psi(e \mapsto \llbracket \lambda x : s. y(x) \rrbracket [y \mapsto e]) \\ &= \Psi(e \mapsto \Psi(d \mapsto \llbracket y(x) \rrbracket [y, x \mapsto e, d])) \\ &= \Psi(e \mapsto \Psi(d \mapsto \Phi(\llbracket y \rrbracket [y, x \mapsto e, d]) (\llbracket x \rrbracket [y, x \mapsto e, d]))) \\ &= \Psi(e \mapsto \Psi(d \mapsto \Phi(e)(d))) \end{aligned}$$

and the value of  $N$  by

$$\llbracket N \rrbracket \emptyset = \Psi(d \mapsto \llbracket y \rrbracket [y \mapsto d]) = \Psi(d \mapsto d).$$

Suppose  $\perp \neq a \in \llbracket \mathbf{o} \rrbracket$ . Then  $\llbracket M \rrbracket \neq \llbracket N \rrbracket$  because  $\Phi \llbracket M \rrbracket \emptyset(a) \neq \Phi \llbracket N \rrbracket \emptyset(a)$ . To see why, first calculate

$$\begin{aligned} \Phi(\llbracket M \rrbracket \emptyset)(a) &= \Phi(\Psi(e \mapsto \Psi(d \mapsto \Phi(e)(d))))(a) \\ &= \Psi(d \mapsto \Phi(a)(d)) \\ &= \Psi(d \mapsto (e \mapsto \perp)(d)) \\ &= \Psi(d \mapsto \perp) \\ &= \perp, \end{aligned}$$

whereas

$$\Phi(\llbracket N \rrbracket \emptyset)(a) = \Phi(\Psi(d \mapsto d))(a) = a \neq \perp.$$

But these two terms are provably equal in the equational theory. Let  $r \equiv s \rightarrow t$ , then by projection

$$\vdash (y : r \triangleright y = y : r),$$

so, by the  $\eta$ -rule,

$$\vdash (y : r \triangleright \lambda x : s. y(x) = y : r).$$

Hence, by the  $\xi$ -rule,

$$\vdash (\triangleright \lambda y : r. \lambda x : s. y(x) = \lambda y : r. y : r \rightarrow r).$$

This is the equation  $M = N$  that is not satisfied by the model. Where does the problem lie here? It is not the  $\eta$ -rule as one might originally suspect: the  $\eta$ -rule is satisfied by the interpretation. The problem is the soundness of the  $\xi$ -rule: the terms  $M$  and  $N$  denote functions, but if they are applied (in the model) to elements of the 'wrong type', then the values may differ.

A construction that solves this problem will be given below, but let us consider how this model is useful even without satisfying the full equational theory for the  $\lambda$ -calculus. The idea presented in [Milner, 1978] is to give a semantic proof of a result such as Theorem 5.1.1. Milner's proof was for the calculus  $ML_0$ , but the idea can be illustrated adequately with PCF. Let us look at the analog of that theorem for PCF with type errors (that is, PCF with a new expression **tyerr** that does not have a type and expanding the operational rules in Table 8 to include rules for type errors as given in Table 11).

To give a fixed-point model of the calculus, we use a domain  $U$  for which there is an isomorphism

$$U \cong \mathbb{T} \oplus \mathbb{N}_\perp \oplus [U \rightarrow U]_\perp \oplus \mathbb{O}. \quad (6.6)$$

In this equation, the operation  $D \mapsto D_\perp$  is the *lift*, which adds a new bottom element to the domain  $D$ . There are obvious maps **up** :  $D \rightarrow D_\perp$  and **down** :  $D_\perp \rightarrow D$  relating this domain to  $D$ . The space  $\mathbb{O}$  is the two-point lattice; let us denote its non-bottom element by **tyerr** to save some notation, since it will be used as the meaning of the term **tyerr**. Let us write  $d : D$  for the injection of element  $d$  into the  $D$  component of the sum. For example,  $n : \mathbb{N}_\perp$  is the injection of  $n \in \mathbb{N}_\perp$  into the second component of  $U$ .

Now, as usual, the semantics of a term  $M$  such that  $H \vdash M : t$  is relative to an  $H$ -environment  $\rho$ .

- $\mathcal{I}[x]\rho = \rho(x)$ .
- $\mathcal{I}[\lambda x : t. M']\rho = \mathbf{up}(f) : [U \rightarrow U]_\perp$  where

$$f(d) = \begin{cases} \mathbf{tyerr} & \text{if } d = \mathbf{tyerr} \\ \mathcal{I}[M']\rho[x \mapsto d] & \text{otherwise.} \end{cases}$$

- $\mathcal{I}[L(N)]\rho = \begin{cases} \mathbf{down}(f)(\mathcal{I}[N]\rho) & \text{if } \mathcal{I}[L]\rho = f : [U \rightarrow U]_\perp \\ \mathbf{tyerr} & \text{otherwise.} \end{cases}$
- $\mathcal{I}[\mu x : t. M']\rho = \mathbf{fix}(d \mapsto \mathcal{I}[M']\rho[x \mapsto d])$ .
- $\mathcal{I}[\mathbf{tyerr}]\rho = \mathbf{tyerr}$ .

The remaining constructs of PCF have a straightforward interpretation that makes the following true:

**Lemma 6.4.4.**

1. For each type  $t$ ,  $\mathcal{I}[t]$  is an inclusive subset.
2. If  $H \vdash M : t$  and  $\rho$  is an  $H$ -environment, then  $\mathcal{I}[M]\rho \in \mathcal{I}[t]$ .
3. If  $M \Downarrow V$ , then  $\mathcal{I}[M] = \mathcal{I}[V]$ .

The point of the lemma is the following:

**Theorem 6.4.5.** If  $M : t$  and  $M \Downarrow V$ , then  $V$  is not **tyerr**.



To see this, note that  $\mathcal{I}[M] = \mathcal{I}[V]$  and  $\mathcal{I}[M]\emptyset \in \mathcal{I}[t]$ . Since  $\mathbf{tyerr} \notin \mathcal{I}[t]$  it follows that  $\mathcal{I}[V]\emptyset \neq \mathbf{tyerr}$ , so it cannot be the case that  $V$  is  $\mathbf{tyerr}$ .

## 6.5 Subtyping as subset inclusion

Let us return now to the topic of a *subtype* as discussed in Section 2.4. In that discussion, the idea that a subtype is a subset was used as an intuition. Using inclusive predicates, it is possible to make this intuition into a formal model. To illustrate this, let us consider a pair of extensions of PCF. The type system considered here is based on ideas in [Reynolds, 1980] and [Cardelli, 1988]; the semantics is basically the one in [Cardelli, 1988].

First we extend PCF to a language that includes records and variants; this extension is called *PCF+* or ‘PCF plus records and variants’. To define its terms, we require a primitive syntax class of *labels*  $l \in \text{Label}$ . Here is its grammar:

$$\begin{array}{ll} x & \in \text{Variable} \\ l & \in \text{Label} \\ t & ::= \mathbf{num} \mid \mathbf{bool} \mid t \rightarrow t \mid \{l : t, \dots, l : t\} \mid [l : t, \dots, l : t] \\ M & ::= \mathbf{0} \mid \mathbf{true} \mid \mathbf{false} \mid \\ & \quad \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{zero?}(M) \mid \mathbf{if } M \mathbf{ then } M \mathbf{ else } M \mid \\ & \quad x \mid \lambda x : t. M \mid MM \mid \mu x : t. M \mid \\ & \quad \{l = M, \dots, l = M\} \mid M.l \mid \\ & \quad [l = M, l : t, \dots, l : t] \mid \mathbf{case } M \mathbf{ of } l \Rightarrow M, \dots, l \Rightarrow M, \end{array}$$

where ellipsis (the notation with three dots) is used to indicate finite lists of pairs in records and variants. The types and terms of PCF+ are of expressions generated by this grammar for which there are no repeated labels in the lists of label/type and label/term pairs. The terms of PCF+ are taken modulo  $\alpha$ -conversion (renaming of bound variables) and in the order in which the fields in records and variants are written. A similar equivalence is assumed for record and variant type expressions.

The typing rules for PCF+ are those for PCF in Tables 4 and 6, together with rules for records and variants given in Table 17. They are quite similar to the rules for products and sums. As with the sum, it is essential to label the variations to ensure that a variant has a unique type. In general we have the following:

**Theorem 6.5.1.** *If  $H \vdash M : s$  and  $H \vdash M : t$ , then  $s \equiv t$ .*

This will not be true of the calculus PCF++ we now consider. PCF++ is the extension of PCF+ in which we allow the use of a subtyping relation between types. The binary relation  $s \leq t$  of subtyping between type expressions  $s$  and  $t$  is defined by the rules in Table 18. It is possible to show that  $\leq$  is a poset on type expressions. The calculus PCF++ is the same as

**Table 17.** Typing rules for records and variants

---

[RecIntro]	$\frac{H \vdash M_1 : t_1 \quad \cdots \quad H \vdash M_n : t_n}{H \vdash \{l_1 = M_1, \dots, l_n = M_n\} : \{l_1 : t_1, \dots, l_n : t_n\}}$
[RecElim]	$\frac{H \vdash M : \{l_1 : t_1, \dots, l_n : t_n\}}{H \vdash M.l_i : t_i}$
[VarIntro]	$\frac{H \vdash M : t}{H \vdash [l = M, l_1 : t_1, \dots, l_n : t_n] : [l : t, l_1 : t_1, \dots, l_n : t_n]}$
[VarElim]	$\frac{\begin{array}{c} H \vdash M : [l_1 : t_1, \dots, l_n : t_n] \\ H \vdash M_1 : t_1 \rightarrow t \quad \cdots \quad H \vdash M_n : t_n \rightarrow t \end{array}}{H \vdash \text{case } M \text{ of } l_1 \Rightarrow M_1 \cdots l_n \Rightarrow M_n : t}$

---

**Table 18.** Rules for subtyping

---

$\text{num} \leq \text{num}$	$\frac{s' \leq s \quad t \leq t'}{s \rightarrow t \leq s' \rightarrow t'}$
$\text{bool} \leq \text{bool}$	
$\frac{s_1 \leq t_1 \quad \cdots \quad s_n \leq t_n}{\{l_1 : s_1, \dots, l_n : s_n, \dots, l_m : s_m\} \leq \{l_1 : t_1, \dots, l_n : t_n\}}$	
$\frac{s_1 \leq t_1 \quad \cdots \quad s_n \leq t_n}{[l_1 : s_1, \dots, l_n : s_n] \leq [l_1 : t_1, \dots, l_n : t_n, \dots, l_m : t_m]}$	

---

PCF+ but with the typing rules of PCF+ extended by the addition of the *subsumption rule*. Since a type can be derived for a term using subsumption that could not be derived without it, it will be essential to distinguish between typing judgements for PCF++ and those of PCF+. Let us write  $\vdash_{\text{sub}}$  for the least relation that contains the relation  $\vdash$  of PCF+ and satisfies

$$[\text{Subsump}] \quad \frac{H \vdash_{\text{sub}} M : s \quad s \leq t}{H \vdash_{\text{sub}} M : t}.$$

The operational semantics of PCF+ and PCF++ is similar to that of PCF but the language is evaluated using call-by-value rather than call-by-name. The rules in Table 8 are used for PCF+ and PCF++ except for the rule:

$$\frac{M \Downarrow \lambda x : s. M' \quad [N/x]M' \Downarrow V}{M(N) \Downarrow V}$$

**Table 19.** Rules for call-by-value evaluation of records and variants

---


$$\frac{M_1 \Downarrow V_1 \quad \cdots \quad M_n \Downarrow V_n}{\{l_1 = M_1, \dots, l_n = M_n\} \Downarrow \{l_1 = V_1, \dots, l_n = V_n\}}$$

$$\frac{M \Downarrow \{l_1 = V_1, \dots, l_n = V_n\}}{M.l_i \Downarrow V_i}$$

$$\frac{M \Downarrow V}{[l = M, \dots] \Downarrow [l = V, \dots]}$$

$$\frac{M \Downarrow [l_i = U, \dots] \quad f_i(U) \Downarrow V}{\text{case } M \text{ of } l_1 \Rightarrow f_1, \dots, l_i \Rightarrow f_i, \dots, l_n \Rightarrow f_n \Downarrow V}$$


---

for evaluation of applications, which is replaced by the rule:

$$\frac{M \Downarrow \lambda x : s. L \quad N \Downarrow U \quad [U/x]L \Downarrow V}{M(N) \Downarrow V}$$

The evaluation of records and variants is given by the rules in Table 19.

A model of PCF++ can be given by extending the inclusive subsets interpretation for PCF. To this end we need a domain similar to the one in Equation (6.6). Let us define

$$U \cong \mathbb{T} \oplus \mathbb{N}_\perp \oplus [U \rightarrow U]_\perp \oplus [\text{Label} \rightarrow U]_\perp \oplus (\text{Label} \times U)_\perp \oplus \mathbb{O}. \quad (6.7)$$

As before, let us write  $d : D$  for the injection of element  $d$  into the  $D$  component of the sum. For example,  $\mathbf{up}(f) : [U \rightarrow U]_\perp$  is the injection of a continuous function  $f : U \rightarrow U$  into the third component of  $U$ . The semantics of types is defined as follows:

- $\mathcal{I}[\{l_1 : t_1, \dots, l_n : t_n\}] = \{r : [\text{Label} \rightarrow U]_\perp \mid \mathbf{down}(r)(l_i) \in \mathcal{I}[t_i] \text{ for each } i = 1, \dots, n\}$
- $\mathcal{I}[l_1 : t_1, \dots, l_n : t_n] = \{e : (\text{Label} \times U)_\perp \mid e = \perp, \text{ or } \mathbf{down}(e) = (l_i, d) \text{ and } d \in \mathcal{I}[t_i]\}$
- $\mathcal{I}[s \rightarrow t] = \{f : [U \rightarrow U]_\perp \mid \mathbf{down}(f)(d) \in \mathcal{I}[t] \text{ for each } d \in \mathcal{I}[s]\}$

The semantics of a term  $M$  such that  $H \vdash_{\text{sub}} M : t$  is relative to an  $H$ -environment  $\rho$ .

- $\mathcal{I}[\{l_1 = M_1, \dots, l_n = M_n\}]\rho = \mathbf{up}(r) : [\text{Label} \rightarrow U]_\perp$  where

$$r(l) = \begin{cases} \mathcal{I}[M_i]\rho & \text{if } l = l_i \\ \mathbf{tyerr} & \text{otherwise} \end{cases}$$

- $\mathcal{I}[M.l]\rho = \begin{cases} \text{down}(f)(l) & \text{if } \mathcal{I}[M]\rho = f : [\text{Label} \rightarrow U]_{\perp} \\ \text{tyerr} & \text{otherwise} \end{cases}$
- $\mathcal{I}[\text{let } l = M, l_1 : t_1, \dots, l_n : t_n] \rho = \text{up}(l, \mathcal{I}[M]\rho) : (\text{Label} \times U)_{\perp}$
- $\mathcal{I}[\text{case } M \text{ of } l_1 \Rightarrow M_1 \cdots l_n \Rightarrow M_n] \rho = d \text{ where}$ 
  - \*  $d = \text{down}(f_i)(e)$  if  $\mathcal{I}[M]\rho = \text{up}(l_i, e) : (\text{Label} \times U)_{\perp}$  and  $\mathcal{I}[M_i]\rho = f_i : [U \rightarrow U]_{\perp}$
  - \*  $d = \perp$  if  $\mathcal{I}[M]\rho = \perp$
  - \*  $d = \text{tyerr}$  otherwise

It is not difficult to check the following property of the interpretation:

**Lemma 6.5.2.** *For each type  $t$  the subset  $\mathcal{I}[t]\rho$  is inclusive.*

Given a suitable choice of the definitions for arithmetic expressions, it is possible to arrange it to be the case that  $\text{tyerr} \notin \mathcal{I}[t]$  for each of the types  $t$  of PCF+. The interpretation also allows us the intuitive liberty of thinking of  $s \leq t$  as meaning that the meaning of  $s$  is a *subset* of the meaning of  $t$ :

**Theorem 6.5.3.** *If  $s \leq t$ , then  $\mathcal{I}[s] \subseteq \mathcal{I}[t]$ .*

The converse of the theorem also holds, if we assume that the solution to Equation 6.7 is an algebraic cpo with a countable basis. Finally, we have the following:

**Theorem 6.5.4.**

1. *If  $H \vdash M : t$  and  $\rho$  is an  $H$ -environment, then  $\mathcal{I}[H \triangleright M : t]\rho \in \mathcal{I}[t]$ .*
2. *If  $M \Downarrow V$ , then  $\mathcal{I}[M] = \mathcal{I}[V]$ .*
3. *If  $M : t$  and  $M \Downarrow V$ , then  $V$  is not **tyerr**.*

## 7 Types as partial equivalence relations

Let us resume the discussion of parametric polymorphism begun earlier by considering some of the type systems used to express this notion. Our goal is to study the distinction between *predicative* and *impredicative* definitions of types and show how the latter can be modeled by interpreting types as equivalence relations on subsets of a universal domain. The system  $\text{ML}_0$  is an example of a predicative system; we begin by demonstrating a set-theoretic model of this system and considering an alternative presentation of its typing rules. This system is then generalized to the best-known impredicative system, the Girard–Reynolds polymorphic  $\lambda$ -calculus. Modeling the impredicativity of the Girard–Reynolds system demands more subtlety; it is shown how this can be done by interpreting types as equivalence relations on subsets of a universal domain.

### 7.1 Sets as a model of $\text{ML}_0$ types

Let us go back now and consider the semantics of  $\text{ML}_0$ . The goal is to provide a model for polymorphic types analogous to the full type frame for

simple types. Recall the syntax of types, type schemes, and terms for the language:

$$\begin{aligned}
 x &\in \text{TermVariable} \\
 a &\in \text{TypeVariable} \\
 t &::= a \mid t \rightarrow t \\
 T &::= t \mid \Pi a. T \\
 M &::= x \mid \lambda x. M \mid MM \mid \text{let } x = M \text{ in } M
 \end{aligned}$$

It will be necessary to have two forms of environment to model the language. Since types may contain variables, we will need the notion of a *type-value environment*,  $\iota$ , which is a function that maps types to semantic domains. An  $H, \iota$ -*environment* is a mapping  $\rho$  that assigns to each  $x \in H$  an element  $\rho(x) \in \llbracket H(x) \rrbracket_\iota$ . For  $\text{ML}_0$  we use sets drawn from a collection obtained by constructing the full type frame.

Let  $X_0$  be any non-empty set; it will serve as the analog of the interpretation of the ground type. For sets  $S, T$ , define  $T^S$  to be the set of functions from  $S$  to  $T$ . Define  $D_0 = \{X_0\}$  and

$$D_{n+1} = \{Y^X \mid X, Y \in D_n\} \cup D_n.$$

The *universe* of our interpretation is the set  $U = \bigcup_{n \in \omega} D_n$ . To model types of  $\text{ML}_0$ , we must interpret type schemes as well as types. Given an operator  $F$  such that  $F(X)$  is a set for each  $X \in U$ , define the *dependent product determined by  $F$*  to be the set  $\Pi_{X \in U} F(X)$  that consists of functions  $\pi$  such that  $\pi(X) \in F(X)$  for each  $X \in U$ . To be more precise about the nature of such maps  $\pi$ , they can be taken as functions with domain  $U$  and range  $\bigcup_{X \in U} F(X)$  satisfying the given constraint that  $\pi(X) \in F(X)$ . The interpretation of types and type schemes can now be given as follows:

- $\llbracket a \rrbracket_\iota = \iota(a)$
- $\llbracket s \rightarrow t \rrbracket_\iota = (\llbracket t \rrbracket_\iota)^{\llbracket s \rrbracket_\iota}$
- $\llbracket \Pi a. T \rrbracket_\iota = \Pi_{X \in U} \llbracket T \rrbracket_\iota[a \mapsto X]$

It is easy to see that  $\llbracket t \rrbracket_\iota$  is an element of  $U$  for each type  $t$  since the universe  $U$  is closed under exponentiation. Note, however, that  $\llbracket \Pi a. T \rrbracket_\iota$  need not be an element of  $U$  despite the fact that type variables  $a$  are mapped to elements of  $U$ . Implicitly we are therefore working with two universes. The first of these,  $U$ , is used for interpreting types, while the second contains sets that can be the interpretations of type schemes. To be more precise, let  $U = V_0$  and, for each  $n \in \omega$ , define  $V_{n+1} = (V_n)^U \cup V_n$ . Then the meaning  $\llbracket \Pi a. T \rrbracket_\rho$  of a type scheme is an element of a second universe  $V = \bigcup_{n \in \omega} V_n$ .

The interpretation of the terms of  $\text{ML}_0$  is more subtle than that of types. Let me write out the equations for the semantics in full and then consider whether they describe a well-defined function.



- Suppose  $H \vdash x : t$  and  $t \leq H(x) \equiv \Pi a_1 \cdots \Pi a_n. s$ . Let  $\sigma$  be a substitution such that  $\sigma(s) \equiv t$ . Letting  $X_i = \llbracket \sigma(a_i) \rrbracket \iota$  for each  $i$ , define  $\llbracket H \triangleright x : t \rrbracket \iota \rho = \rho(x)(X_1) \cdots (X_n)$ .
- $\llbracket H \triangleright \lambda x. M : s \rightarrow t \rrbracket \iota \rho$  is the function from  $\llbracket s \rrbracket \iota$  to  $\llbracket t \rrbracket \iota$  defined by  $d \mapsto \llbracket H, x : s \triangleright M : t \rrbracket \iota \rho[x \mapsto d]$ .
- $\llbracket H \triangleright M(N) : t \rrbracket \iota \rho = (\llbracket H \triangleright M : s \rightarrow t \rrbracket \iota \rho)(\llbracket H \triangleright N : s \rrbracket \iota \rho)$ .
- Suppose  $\text{close}(H; s) = \Pi a_1 \cdots \Pi a_n. s$  and  $H \vdash M : s$ . Define  $\pi \in \llbracket \text{close}(H; s) \rrbracket \iota$  by

$$\pi(X_1) \cdots (X_n) = \llbracket H \triangleright M : s \rrbracket (\iota[a_1, \dots, a_n \mapsto X_1, \dots, X_n]) \rho.$$

Then  $\llbracket H \triangleright \text{let } x = M \text{ in } N : t \rrbracket \iota \rho = \llbracket H, x : \text{close}(H; s) \triangleright N : t \rrbracket \iota \rho[x \mapsto \pi]$ .

The primary question about the sense of this definition concerns whether the type-value environment  $\iota' = \iota[a_1, \dots, a_n \mapsto X_1, \dots, X_n]$  in the semantic equation for the **let** construct is compatible with the environment  $\rho$ ; that is, whether  $\rho$  is an  $H, \iota'$ -environment. This question is resolved by recalling that none of the type variables  $a_i$  is in  $\text{Ftv}(H)$  and by noting the following:

**Lemma 7.1.1.** *If  $\rho$  is an  $H, \iota$ -environment and  $a \notin \text{Ftv}(H)$ , then it is also an  $H, \iota[a \mapsto X]$ -environment.*

This follows from the fact that  $\llbracket t \rrbracket \iota[a \mapsto X] = \llbracket t \rrbracket \iota$  if  $a$  is not free in  $t$ .

## 7.2 Another typing system for $\text{ML}_0$ .

In light of the semantics we just gave, the type system we have been using for  $\text{ML}_0$  appears to be slightly indirect in some ways. In the rule [Let], for instance, the meaning of the term  $M$  in **let**  $x = M$  **in**  $N$  is calculated, and then a ‘parameterized’ version of its meaning is bound to  $x$  in the environment. Similarly, the meaning of a variable is drawn from the environment and then instantiated to the type assigned to  $x$ . Permitting judgements of the form  $H \vdash M : T$ , where  $T$  is a type scheme, together with rules for generalization and instantiation, might lead a more elementary system. It is indeed possible to reformulate the typing system for  $\text{ML}_0$  in this way. The rules for deriving such judgements appear in Table 20. The rules for abstraction and application remain unchanged, but the rules for projections and **let** constructs now reflect the more general form of judgement in this system. In the new system, the projection rule looks more or less the way it does in most of the systems we have considered rather than having the somewhat different form it has in Table 13. The ‘close’ operator is no longer used in the rule for **let** since the term  $M$  in the hypothesis may be given a type scheme rather than a type that must be generalized on variables not in  $H$ . But the real difference in the two systems lies in the presence of rules for introduction and elimination of  $\Pi$  bindings. One

**Table 20.** Typing rules for  $ML_0$  with  $\Pi$  introduction and elimination

---

[Proj]	$\frac{x : T \in H}{H \vdash x : T}$
[Abs] <sup>-</sup>	$\frac{H, x : s \vdash M : t}{H \vdash \lambda x. M : s \rightarrow t}$
[Appl]	$\frac{H \vdash M : s \rightarrow t \quad H \vdash N : s}{H \vdash M(N) : t}$
[Let]	$\frac{H \vdash M : T \quad H, x : T \vdash N : t}{H \vdash \text{let } x = M \text{ in } N : t}$
[ $\Pi$ -Intro] <sup>-</sup>	$\frac{H \vdash M : T \quad a \notin \text{Ftv}(H)}{H \vdash M : \Pi a. T}$
[ $\Pi$ -Elim] <sup>-</sup>	$\frac{H \vdash M : \Pi a. T}{H \vdash M : [t/a]T}$

---

particular difference made by the addition of these rules is the fact that the derivation of a judgement  $H \vdash M : T$  is not uniquely determined by  $H, M, T$ . There will, in fact, be many (superficially) distinct proofs of any such judgement obtained by alternating the application of rules [ $\Pi$ -Intro]<sup>-</sup> and [ $\Pi$ -Elim]<sup>-</sup>. Nevertheless, it is possible to show that two systems are essentially the same on judgements of types.

**Proposition 7.2.1.** *Let  $H$  be a type assignment,  $M$  a term, and  $t$  a type.*

1. *If  $H \Vdash M : t$ , then  $H \vdash M : t$ .*
2. *If  $H \vdash M : T$  and  $t \leq T$ , then  $H \Vdash M : t$ .*

### 7.3 The polymorphic $\lambda$ -calculus

In many of the calculi we have considered, type annotations in terms were used to force each typeable term to have a unique type. The rules tagged with a superscript minus sign in the typing system for  $ML_0$  described in Table 20 cause this property to fail. To recover it, we might place type tags on the  $\lambda$ -bound variables, but the rules [ $\Pi$ -Intro]<sup>-</sup>, [ $\Pi$ -Elim]<sup>-</sup> would still pose a problem. In effect, terms must contain some indication about the generalization and instantiation of type variables if their types are to be uniquely determined. Let us now consider an important generalization of  $ML_0$  that has explicit abstraction and application for type variables in terms. The system is sometimes called the *Girard-Reynolds* polymorphic  $\lambda$ -calculus, since the system was discovered independently by Girard [1972] (who was working on a proof-theoretic problem) and by Reynolds [Reynolds, 1974]

(who was interested in programming language design). With the possible exception of ‘ML polymorphism’, it is the best-known polymorphic type system, so it is most often simply called the *polymorphic  $\lambda$ -calculus*. The reader is referred to the survey paper [Scedrov, 1990] for a fuller discussion of polymorphic  $\lambda$ -calculus, including references and historical background. The terms of the language are given as follows:

$$\begin{aligned}
 x &\in \text{TermVariable} \\
 a &\in \text{TypeVariable} \\
 t &::= t \rightarrow t \mid a \mid \Pi a. t \\
 M &::= x \mid \lambda x : t. M \mid M(M) \mid \Lambda a. M \mid M\{t\}
 \end{aligned}$$

A term of the form  $\Lambda a. M$  is called a *type abstraction*, and one of the form  $M\{t\}$  is called a *type application*. Types of the form  $\Pi a. t$  are called  *$\Pi$ -types*, and the type variable  $a$  is bound in  $\Pi a. t$  by the  $\Pi$ -quantification. The following clauses define the free type variables for types and terms:

- $\text{Ftv}(a) = a$
- $\text{Ftv}(s \rightarrow t) = \text{Ftv}(s) \cup \text{Ftv}(t)$
- $\text{Ftv}(\Pi a. t) = \text{Ftv}(t) - \{a\}$
- $\text{Ftv}(x) = \emptyset$
- $\text{Ftv}(\lambda x : t. M) = \text{Ftv}(t) \cup \text{Ftv}(M)$
- $\text{Ftv}(M(N)) = \text{Ftv}(M) \cup \text{Ftv}(N)$
- $\text{Ftv}(\Lambda a. M) = \text{Ftv}(M) - \{a\}$
- $\text{Ftv}(M\{t\}) = \text{Ftv}(M) \cup \text{Ftv}(t)$

For an assignment  $H$ , the set of free type variables  $\text{Ftv}(H)$  in  $H$  is the union of the free type variables in  $H(x)$  for each  $x \in H$ . Substitution for both types and terms must respect bindings in the sense that no free variable of the term being substituted can be captured by a binding in the term into which the substitution is made.

The types for terms of the polymorphic  $\lambda$ -calculus may be built using type variables. For example,  $\lambda y : a. \lambda x : a \rightarrow b. x(y)$  is a well-typed term with type  $a \rightarrow (a \rightarrow b) \rightarrow b$ . However, unlike the  $\text{ML}_0$  systems, the polymorphic  $\lambda$ -calculus allows type variables to be explicitly abstracted in terms. For example, the term

$$M \equiv \Lambda a. \Lambda b. \lambda y : a. \lambda x : a \rightarrow b. x(y)$$

has the type  $\Pi a. \Pi b. a \rightarrow (a \rightarrow b) \rightarrow b$ . It is possible to instantiate the abstracted type variables through a form of application. Given types  $s$  and  $t$ , for example,  $M\{s\}\{t\}$  is equivalent to the term  $\lambda y : s. \lambda x : s \rightarrow t. x(y)$ . This latter term has the type  $s \rightarrow (s \rightarrow t) \rightarrow t$ .

The precise typing rules for the polymorphic  $\lambda$ -calculus are those in Table 4 together with the two rules that appear in Table 21. Of course,

the rules in Table 4 must be understood as applying to all of the terms of the polymorphic calculus as given in the grammar for the language (and not just to the terms of the simply-typed calculus). As with earlier calculi,

**Table 21.** Typing rules for the polymorphic  $\lambda$ -calculus

[II-Intro]	$\frac{H \vdash M : s \quad a \notin \text{Ftv}(H)}{H \vdash \Lambda a. M : \Pi a. s}$
[II-Elim]	$\frac{H \vdash M : \Pi a. s}{H \vdash M\{t\} : [t/a]s}$

the type tags on the bound variables ensure the following:

**Lemma 7.3.1.** *For any type assignment  $H$ , term  $M$ , and type expressions  $s, t$ , if  $H \vdash M : s$  and  $H \vdash M : t$  then  $s \equiv t$ .*

The virtue of the polymorphic  $\lambda$ -calculus is that it can be used to express general algorithms in a clear way. For example, let us return to the problem illustrated earlier by the program

```
(define applyto
  (lambda (f) (cons (f 3) (f "hi"))))
```

The function `applyto` takes a function as an argument and applies it to each of the components of a given pair, returning a pair as a result. Here is a similar program written in the polymorphic  $\lambda$ -calculus extended with pairs:

$$\Lambda a. \lambda f : (\Pi b. b \rightarrow a). (f\{\text{int}\}(M), f\{\text{string}\}(N)).$$

The types must be explicitly instantiated as part of the application, but the program is more general than any that can be written in  $\text{ML}_0$ . More convincing programming examples can be given, but this shows that the phenomenon arises quite naturally.

The equational rules for the pure polymorphic  $\lambda$ -calculus are those in Table ?? together with the rules that appear in Table 22 modulo the theory  $T$  that appears on the left-hand sides of the turnstiles.<sup>1</sup> The new rules  $\{\text{TypeCong}\}$  and  $\{\text{Type}\xi\}$  assert that type application and type abstraction are congruences. The most fundamental new rules are the type-level analogs  $\{\text{Type}\beta\}$  and  $\{\text{Type}\eta\}$  of the  $\beta$  and  $\eta$  rules.

<sup>1</sup>We could also define the polymorphic  $\lambda$ -calculus more generally relative to a theory  $T$ , but the discussion here is based on using the empty theory.

**Table 22.** Equational rules for the polymorphic  $\lambda$ -calculus

---

$\{\text{TypeCong}\}$	$\frac{\vdash (H \triangleright M = N : \Pi a. t)}{\vdash (H \triangleright M\{s\} = N\{s\} : [s/a]t)}$
$\{\text{Type } \xi\}$	$\frac{\vdash (H \triangleright M = N : t)}{\vdash (H \triangleright \Lambda a. M = \Lambda a. N : \Pi a. t)}$
$\{\text{Type } \beta\}$	$\frac{H \vdash M : t}{\vdash (H \triangleright (\Lambda a. M)\{s\} = [s/a]M : [s/a]t)}$
$\{\text{Type } \eta\}$	$\frac{H \vdash M : \Pi a. t}{\vdash (H \triangleright \Lambda a. M\{a\} = M : \Pi a. t)}$

---

Restrictions:

- In  $\{\text{Type } \xi\}$ , there is no free occurrence of  $a$  in the type of a variable in  $H$ .
- In  $\{\text{Type } \eta\}$ , the type variable  $a$  does not appear free in  $H$  or  $M$ .

---

## 7.4 Sets as a model of polymorphic types?

The interpretation of the polymorphic  $\lambda$ -calculus has been one of the most serious challenges in the semantics of programming languages. Of course, it is possible to construct a term model for the calculus as we did earlier for the simply-typed  $\lambda$ -calculus. But finding a model analogous to the full type frame is much harder. To appreciate the primary reason for this difficulty, let us attempt to provide such a model by partial analogy to the one we used for  $\text{ML}_0$ . We will need the notion of a *type-value environment*  $\iota$  that maps type variables to semantic interpretations as sets. The interpretation  $\llbracket t \rrbracket$  of a type  $t$  is a function that takes type assignments indicating the meanings of free variables of  $t$  into sets. As with the full type frame, we define  $\llbracket s \rightarrow t \rrbracket \iota = \llbracket s \rrbracket \iota \rightarrow \llbracket t \rrbracket \iota$  where the arrow on the right is the full function space operator. The central question is, how do we interpret  $\Pi a. t$ ? Let us naively take this to be a product of sets indexed over sets; an element of  $\llbracket \Pi a. t \rrbracket \iota$  is a function that associates with each set  $X$  an element of the set  $\llbracket t \rrbracket \iota[a \mapsto X]$ . Such a function is called a *section* of the *indexed family*  $X \mapsto \llbracket t \rrbracket \iota[a \mapsto X]$ . It can improve the readability of expressions involving sections to write the application of a section to a set with the argument as a subscript. So, for example, if  $\pi$  is a section of  $X \mapsto \llbracket t \rrbracket \iota[a \mapsto X]$ , then  $\pi_X \in \llbracket t \rrbracket \iota[a \mapsto X]$ . To provide the semantic interpretation for terms, we will also want to know that a form of substitution lemma holds for types:  $\llbracket [s/a]t \rrbracket \iota = \llbracket t \rrbracket \iota[a \mapsto \llbracket s \rrbracket \iota]$ .

Given a type-value environment  $\iota$  and a type assignment  $H$ , let us say that  $\rho$  is an  $\iota, H$ -environment if  $\rho(x) \in \llbracket H(x) \rrbracket \iota$  for each  $x \in H$ . If



$H \vdash M : t$ , then the interpretation  $\llbracket H \triangleright M : t \rrbracket$  is a function that takes a type-value environment  $\iota$  and an  $\iota, H$ -environment as an argument and returns a value in  $\llbracket t \rrbracket \iota$ . It sometimes helps to drop the type information in the interpreted expression and write  $\llbracket M \rrbracket \iota \rho$  to save clutter when the types are clear enough from context.

The interpretation of terms is now defined by induction on their structure. The interpretation of an abstraction  $\llbracket H \triangleright \lambda x : s. M : s \rightarrow t \rrbracket \iota \rho$  over term variables is the function from  $\llbracket s \rrbracket \iota$  to  $\llbracket t \rrbracket \iota$  defined by

$$d \mapsto \llbracket H, x : s \triangleright M : t \rrbracket \iota (\rho[x \mapsto d]).$$

On the other hand, the interpretation of the application of a term to a term is given by the usual application of a function to its argument:  $\llbracket M(N) \rrbracket \iota \rho = (\llbracket M \rrbracket \iota \rho)(\llbracket N \rrbracket \iota \rho)$ . In considering the interpretation of the application of a term  $M : \Pi a. s$  to a type  $t$ , recall that  $\llbracket H \triangleright M : \Pi a. s \rrbracket \iota \rho$  is a section of the indexed family  $X \mapsto \llbracket s \rrbracket \iota[a \mapsto X]$ . We take  $\llbracket H \triangleright M \{t\} : [t/a]s \rrbracket \iota \rho = (\llbracket H \triangleright M : \Pi a. s \rrbracket \iota \rho)_X$  where  $X = \llbracket t \rrbracket \iota$ . This squares with the claim that  $\llbracket [t/a]s \rrbracket \iota = \llbracket s \rrbracket \iota[a \mapsto \llbracket t \rrbracket \iota]$ . Now, finally, the meaning of a type abstraction  $\llbracket H \triangleright \Lambda a. M : \Pi a. t \rrbracket \iota \rho$  is a section of the indexed family  $X \mapsto \llbracket t \rrbracket \iota[a \mapsto X]$  given by  $X \mapsto \llbracket H \triangleright M : t \rrbracket (\iota[a \mapsto X]) \rho$ . One must show that  $\rho$  is an  $\iota[a \mapsto X]$ ,  $H$ -value environment, but this follows from the restriction in [II-Intro] that says the type variable  $a$  does not appear in  $H$ .

The semantics just sketched is sufficiently simple and convincing that something like it was actually used in early discussions of the semantics of the calculus [Reynolds, 1983]. As it stands, however, there is a problem with the interpretation of types. A type is presumably a function from type-value environments to sets. But consider a type like  $\Pi a. a$ , which we have naively interpreted as the family of sections of the indexed family  $X \mapsto X$ . In other words,  $\llbracket \Pi a. a \rrbracket \iota$  is the ‘product’ of *all* sets. We must assume that this product is itself a set, because this is needed to make our interpretation work. Consider, for instance, the term  $M = (\Lambda a. \lambda x : a. x)(\Pi a. a)$ . The term  $\Lambda a. \lambda x : a. x$  is interpreted as a section over all sets and  $M$  is interpreted as the application of this section to the meaning of  $\Pi a. a$ .

This leads us to a foundational question concerning what collections are considered to be sets. One of the crucial discoveries of logicians in the late nineteenth and early twentieth centuries was the fact that care must be taken in how collections of entities are formed if troubling paradoxes are to be avoided. Such paradoxes caused intricate and carefully constructed theories of the foundations of mathematics to collapse into nonsense. Perhaps the best-known and most important of these paradoxes is *Russell’s paradox*, which is named after the philosopher and logician Bertrand Russell. It can be described quite simply as follows. Let us assume that any property at all can be used to define a collection of entities. Define  $\mathcal{X}$  to be the

collection of all collections  $X$  having the property that  $X$  is not an element of  $X$ . This seems clear enough since we (think we) know what it means for an entity to be part of a collection. Let us therefore ask whether  $\mathcal{X}$  is in  $\mathcal{X}$  or not. Well, if it is, then it has the property common to all elements of  $\mathcal{X}$ , that of not being a member of itself. This is a contradiction, since we had postulated that  $\mathcal{X}$  was a member of itself. Suppose, on the other hand, that  $\mathcal{X}$  is *not* a member of itself. Then this is a contradiction as well, since we defined  $\mathcal{X}$  to be those collections having exactly this property.

Returning now to the polymorphic  $\lambda$ -calculus and our problem with its interpretation, we must avoid a transgression into Russell's paradox. Technically the problem is that the restrictions placed on the formation of sets makes it impossible to view the product of all sets as itself a set. However, the underlying phenomenon here was recognized by Russell and by the mathematician and philosopher of science Henri Poincaré in a property he called 'impredicativity'. If a set  $\mathcal{X}$  and an entity  $X$  are defined so that  $X$  is a member of  $\mathcal{X}$  but is defined only by reference to  $\mathcal{X}$ , then the definition of  $\mathcal{X}$  or  $X$  is said to be *impredicative*. Clearly this is the case for Russell's paradox, but it also applies to the class  $\mathcal{X}$  of semantic domains that we are attempting to use for modeling the types of the polymorphic  $\lambda$ -calculus and the domain  $X$  that is to serve as the interpretation for  $\Pi a. a$ . That this problem has no resolution when one is dealing with *sets* was shown by Reynolds [1984] (a more refined treatment appears in [Reynolds and Plotkin, 1990]); thus we must look for another class of semantics domains with which to interpret our types.

## 7.5 Simple types as PERs

Recall the problem cited earlier with the use of inclusive predicates to model types: namely that the  $\xi$ -rule is not satisfied. An approach to solving this problem is to use an equivalence relation on subsets of the universal domain to induce the needed equalities. In particular:

**Definition 7.5.1.** Let  $A$  be a set. A *partial equivalence relation (PER)* on  $A$  is a relation  $R \subseteq A \times A$  that is transitive and symmetric. The *domain* of a partial equivalence relation  $R$  is the set  $\text{dom}(R) = \{a \in A \mid a R a\}$ .

We write  $a R b$  to mean that  $(a, b) \in R$ . Note that if  $R$  is a PER on a set  $A$  and  $a R b$  for any  $a, b \in A$ , then  $a$  and  $b$  are in the domain of  $A$  by the axioms.

Now, suppose we are given a model of the untyped  $\lambda$ -calculus. Say  $\Phi : U \rightarrow [U \rightarrow U]$  and  $\Psi : [U \rightarrow U] \rightarrow U$  satisfy  $\Phi \circ \Psi = \text{id}_{[U \rightarrow U]}$ . Let  $X$  be any PER on  $U$ . Take  $\mathcal{P}[\mathbf{o}] = X$  and define PER interpretations for types by structural induction as follows. Suppose  $\mathcal{P}[s]$  and  $\mathcal{P}[t]$  are PERs, then

$$\begin{aligned}
 & f(\mathcal{P}[s \rightarrow t])g \\
 & \text{iff} \\
 & \text{for each } d \text{ and } e, d(\mathcal{P}[s])e \text{ implies } \Phi(f)(d)(\mathcal{P}[t])\Phi(g)(e).
 \end{aligned} \tag{7.1}$$

It is easy to check that each of these relations is a partial equivalence. To see how they interpret terms, we use a semantic function that gives untyped meaning to typed terms. Given a term  $M$  of the simply-typed  $\lambda$ -calculus, let  $\mathcal{U}[M]$  be the meaning of the untyped  $\lambda$ -term  $\text{erase}(M)$  in  $U$  based on the pair  $\Phi, \Psi$ . Given a PER  $R$  on  $U$  and an element  $d \in \text{dom}(R)$ , let  $[d]_R$  be the equivalence class of  $d$  relative to  $R$ , that is,  $[d]_R = \{e \mid d R e\}$ .

Let  $M$  be a term of the typed  $\lambda$ -calculus such that  $H \vdash M : t$ . The meaning of  $M$  is given relative to a function  $\rho$  from variables  $x \in H$  into  $U$  such that  $\rho(x)$  is in the domain of the relation  $\mathcal{P}[H(x)]$  for each  $x \in H$ . Such a function is called an  $H$ -environment for the PER interpretation. Now, the interpretation for the term  $M$  is quite simple:

$$\mathcal{P}[H \triangleright M : t]\rho = [\mathcal{U}[M]\rho]_{\mathcal{P}[t]}$$

There are two basic facts to be proved about this interpretation. First, if  $M$  has type  $t$  then the interpretation of  $M$  is in the domain of the relation  $\mathcal{P}[t]$ . Second, all of the equational rules of the typed  $\lambda$ -calculus are satisfied. Establishing these properties involves proving slightly more general facts. Given  $H$ -environments  $\rho$  and  $\theta$ , define  $\rho \sim_H \theta$  if, for each  $x \in H$ ,  $\rho(x) \mathcal{P}[H(x)] \theta(x)$ .

**Lemma 7.5.2.** *Suppose  $H \vdash M : t$ . If  $\rho$  and  $\theta$  are  $H$ -environments and  $\rho \sim_H \theta$ , then  $(\mathcal{U}[M]\rho) \mathcal{P}[t] (\mathcal{U}[M]\theta)$ .*

**Proof.** The proof is by induction on the structure of  $M$ . If  $M \equiv x$ , then  $x \in H$ ,  $t \equiv H(x)$ , and  $\rho \sim \theta$  implies the desired conclusion.

Case  $M \equiv \lambda x : u. M'$  where  $t \equiv u \rightarrow v$ . Suppose  $d \mathcal{P}[u] e$ . Then

$$\Phi(\mathcal{U}[\lambda x : u. M']\rho)(d) = \Phi(\Psi(d \mapsto \mathcal{U}[M']\rho[x \mapsto d]))(d) = \mathcal{U}[M']\rho[x \mapsto d],$$

and, similarly,  $\Phi(\mathcal{U}[\lambda x : u. M']\theta)(e) = \mathcal{U}[M']\theta[x \mapsto e]$ . Now,

$$(\mathcal{U}[M']\rho[x \mapsto d]) \mathcal{P}[v] (\mathcal{U}[\lambda x : u. M']\theta[x \mapsto e])$$

by the inductive hypothesis. The desired conclusion therefore follows from the definition of  $\mathcal{P}[t]$ .

Case  $M \equiv L(N)$  where  $H \vdash L : s \rightarrow t$  and  $H \vdash N : s$ . By the inductive hypothesis for  $N$ ,  $(\mathcal{U}[N]\rho) \mathcal{P}[s] (\mathcal{U}[N]\theta)$ . So, by the inductive hypothesis for  $L$ ,  $d \mathcal{P}[t] e$  where  $d = \Phi(\mathcal{U}[L]\rho)(\mathcal{U}[N]\rho) = \mathcal{U}[M]\rho$  and  $e = \Phi(\mathcal{U}[L]\theta)(\mathcal{U}[N]\theta) = \mathcal{U}[M]\theta$ . ■

**Corollary 7.5.3.** *If  $H \vdash M : t$  and  $\rho$  is an  $H$ -environment, then  $\mathcal{P}[H \triangleright M : t]\rho$  is in the domain of the relation  $\mathcal{P}[t]$ .*

**Lemma 7.5.4.** *If  $\vdash (H \triangleright M' = N' : t')$  and  $\rho \sim_H \theta$  are  $H$ -environments, then*

$$\mathcal{P}[H \triangleright M' : t']\rho = \mathcal{P}[H \triangleright N' : t']\theta.$$

**Proof.** The proof is by induction on the height of the derivation of the judgement  $\vdash (H \triangleright M' = N' : t')$ . By way of illustration, let us consider the case in which the last step of the proof is an instance of the  $\xi$ -rule. Suppose the last step of the derivation is an instance of

$$\{\xi\} \quad \frac{T \vdash (H, x : s \triangleright M = N : t)}{T \vdash (H \triangleright \lambda x : s. M = \lambda x : s. N : s \rightarrow t)}$$

where  $M' \equiv \lambda x : s. M$  and  $N' \equiv \lambda x : s. N$  and  $t' \equiv s \rightarrow t$ . By the inductive hypothesis,  $(\mathcal{U}[M]\rho[x \mapsto d]) \mathcal{P}[t]$   $(\mathcal{U}[N]\theta[x \mapsto e])$  whenever  $e \mathcal{P}[s]$   $d$ . Hence  $(\mathcal{U}[\lambda x. M]\rho) \mathcal{P}[s \rightarrow t]$   $(\mathcal{U}[\lambda x. N]\theta)$  by the definition of the PER  $\mathcal{P}[s \rightarrow t]$ . Thus  $\mathcal{P}[H \triangleright \lambda x : s. M : s \rightarrow t]\rho = \mathcal{P}[H \triangleright \lambda x : s. N : s \rightarrow t]\theta$ . ■

**Corollary 7.5.5.** *If  $\vdash (H \triangleright M = N : t)$ , then  $\mathcal{P}[H \triangleright M : t] = \mathcal{P}[H \triangleright N : t]$ .*

## 7.6 PERs as a model of polymorphic types

We have seen that if we are given a model  $(U, \Phi, \Psi)$  of the untyped  $\lambda$ -calculus, then PERs over  $U$  can be used to interpret the types of the simply-typed  $\lambda$ -calculus; now we consider how PERs can be used to interpret types of the polymorphic  $\lambda$ -calculus. To make this extension, we define the meaning of a type relative to a type-value environment that maps type variables to PERs. To interpret  $\Pi a. t$  as an indexed family over PERs, let  $\iota$  be a type-value environment that has all of the free type variables of  $\Pi a. t$  in its domain. We want to define its meaning to be the ‘product’ of the relations  $\mathcal{P}[t]\iota[a \mapsto R]$  as  $R$  ranges over the PERs over  $U$ . Given  $x, y \in U$ , this says that  $x$  and  $y$  are related modulo  $\mathcal{P}[t]\iota[a \mapsto R]$  for each  $R$ . That is,

$$\mathcal{P}[\Pi a. t]\iota = \bigcap_{R \in \text{PER}} \mathcal{P}[t]\iota[a \mapsto R]. \quad (7.2)$$

This defines a partial equivalence relation because the intersection of PERs is a PER. Notice the role of impredicativity in Equation (7.2) where the intersection ranges over the class of all PERs. Of course, the PER that interprets  $\Pi a. t$  itself is in this collection, but there is no problem with existence in this case, because the intersection of a set of sets is again a set. The interpretation of function spaces is given same as before in (7.1).

The *erasure* of a term of the polymorphic  $\lambda$ -calculus is defined by induction as follows:

- $\text{erase}(x) \equiv x$



- $\text{erase}(\lambda x : t. M) \equiv \lambda x. \text{erase}(M)$
- $\text{erase}(M(N)) \equiv (\text{erase}(M))(\text{erase}(N))$
- $\text{erase}(\Lambda a. M) \equiv \text{erase}(M)$
- $\text{erase}(M\{t\}) \equiv \text{erase}(M)$

The meaning of a term is defined using the meaning, as an untyped term, of its erasure. We assume that a retraction from  $U$  onto  $[U \rightarrow U]$  is given and define  $\mathcal{U}[M]$  to be the meaning of the untyped  $\lambda$ -term  $\text{erase}(M)$  in  $U$ . As before, given a PER  $R$  on  $U$  and an element  $d \in \text{dom}(R)$ , let  $[d]_R$  be the equivalence class of  $d$  relative to  $R$ , that is,  $[d]_R = \{e \mid d R e\}$ . Let  $M$  be a term of the polymorphic  $\lambda$ -calculus such that  $H \vdash M : t$ . The meaning of  $M$  is given relative to a type-value environment  $\iota$  and a function  $\rho$  from variables  $x \in H$  into  $U$  such that  $\rho(x)$  is in the domain of the relation  $\mathcal{P}[H(x)]\iota$ . The interpretation of the term  $M$  is given by

$$\mathcal{P}[M]\iota\rho = [\mathcal{U}[M]\rho]_{\mathcal{P}[t]\iota}.$$

To complete the demonstration that this defines a model, it must be shown that if a term  $M$  has type  $t$ , then the interpretation of  $M$  relative to an  $H, \iota$ -environment is in the domain of the relation  $\mathcal{P}[t]\iota$ , and that the equational rules of the polymorphic  $\lambda$ -calculus are satisfied. The treatment follows the general pattern of the argument for interpretation of simple types. We start with the following basic lemma:

**Lemma 7.6.1.**  $\mathcal{P}[[s/a]t]\iota = \mathcal{P}[t]\iota[a \mapsto \mathcal{P}[s]\iota]$ .

**Lemma 7.6.2.** Suppose  $H \vdash M : t$  and  $\rho, \theta$  are  $H, \iota$ -environments such that

$$\rho(x) (\mathcal{P}[H(x)]\iota) \theta(x)$$

for each  $x \in H$ . Then

$$(\mathcal{U}[M]\rho) (\mathcal{P}[t]\iota) (\mathcal{U}[M]\theta).$$

**Proof.** The proof is by induction on the structure of  $M$ . Let us look at the cases for type abstraction and application. If  $M \equiv \Lambda a. M' : \Pi a. t'$ , then

$$(\mathcal{U}[M']\rho) (\mathcal{P}[t']\iota[a \mapsto R]) (\mathcal{U}[M']\theta)$$

for any PER  $R$  by the inductive hypothesis. Since  $\text{erase}(M') \equiv \text{erase}(M)$  and the interpretation of  $\Pi a. t'$  is the intersection of PERs of the form  $\mathcal{P}[t']\iota[a \mapsto R]$ , we must have

$$(\mathcal{U}[M]\rho) (\mathcal{P}[\Pi a. t']\iota) (\mathcal{U}[M]\theta).$$

Suppose now that  $M \equiv M'\{s\}$ . Then  $H \vdash M' : \Pi a. t'$  and  $t \equiv [s/a]t'$ . Applying the inductive hypothesis to  $M'$ , we have



$$(\mathcal{U}[\![M']\!]\rho) \left( \bigcap_{R \in PER} \mathcal{P}[\![t']\!]\iota[a \mapsto R] \right) (\mathcal{U}[\![M']\!]\theta)$$

and therefore, in particular,

$$(\mathcal{U}[\![M']\!]\rho) (\mathcal{P}[\![t']\!]\iota[a \mapsto \mathcal{P}[\![s]\!]\iota]) (\mathcal{U}[\![M']\!]\theta)$$

Now  $\text{erase}(M') \equiv \text{erase}(M)$  so, by Lemma 7.6.1,

$$(\mathcal{U}[\![M]\!]\rho) (\mathcal{P}[\![s/a]t'\!]\iota) (\mathcal{U}[\![M]\!]\theta).$$

**Corollary 7.6.3.** *If  $H \vdash M : t$  and  $\rho$  is an  $H, \iota$ -environment, then  $\mathcal{U}[H \triangleright M : t]\iota\rho$  is in the domain of the relation  $\mathcal{P}[t]\iota$ .*

**Lemma 7.6.4.** *If  $\vdash (H \triangleright M' = N' : t')$  and  $\rho, \theta$  are  $H, \iota$ -environments such that  $\rho(x) (\mathcal{P}[H(x)]\iota) \theta(x)$  for each  $x \in H$ , then*

$$\mathcal{P}[H \triangleright M' : t']\iota\rho = \mathcal{P}[H \triangleright N' : t']\iota\theta.$$

**Corollary 7.6.5.** *If  $\vdash (H \triangleright M = N : t)$ , then  $\mathcal{P}[H \triangleright M : t] = \mathcal{P}[H \triangleright N : t]$ .*

## 8 Conclusion

A more detailed treatment of the topics in this chapter, including complete proofs of most of the theorems, can be found in [Gunter, 1992]; a great deal of further material on the semantics of types can be found in other chapters of this handbook. *There is much more that can be said about each of the models described in the sections here:* for example, since seminal work of Bruce and Longo [Bruce and Longo, 1990], PER's have been quite successful as a model of *subtypes* as well as parametric polymorphism and, as one can glean from the references in [Scedrov, 1990], we have only just scratched the surface of what can be said about PER's as a model of parametric polymorphism. *There are other models besides the ones that have been covered here:* a whole subject of the semantics of types as *categorical objects* has been omitted (see [Poigné, 1992]) and there is a rapidly evolving theory of types as *games* [Abramsky et al., 1993]. And, particularly, *there are hosts of type systems* that have not been discussed in this chapter. One large category of these only touched upon here is that of types for *object-oriented* programming languages. The reader can pursue this topic further through [Gunter and Mitchell, 1994; Palsberg and Schwartzback, 1993] and the references there. A second is the semantics of recursive types, whose importance was hinted at in Section 2 but not discussed in any detail in the remainder of the chapter; the

reader can pursue the topic further by consulting [Gunter, 1992]. A third important class of type systems is those that are used for *modules*; two references that can be used as a starting point are [Harper and Mitchell, 1993] and [Moggi, 1991].

I would like to acknowledge Samson Abramsky for encouraging me to undertake this discussion of the semantics of types. I also thank Sandip Biswas, Roy Crole, and Narciso Martí-Oliet for their help in proof-reading drafts of the work.

## References

- [Abelson and Sussman, 1985] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.
- [Abramsky and Jung, 1994] S. Abramsky and A. Jung. Domains. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science*, pages 1–190. Oxford University Press, 1994.
- [Abramsky *et al.*, 1993] S. Abramsky, R. Jagadeesan, and P. Malacaria. Games and full abstraction for PCF II: Second preliminary announcement. Manuscript, 1993.
- [Barendregt, 1992] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science, Volume 2 Background: Computational Structures*, pages 117–310. Oxford University Press, 1992.
- [Berry, 1978] G. Berry. Stable models of typed  $\lambda$ -calculus. In *International Colloquium on Automata, Languages and Programs*, pages 72–89. *Lecture Notes in Computer Science* vol. 62, Springer, 1978.
- [Berry, 1979] G. Berry. *Modèles Complètement Adéquats et Stables des Lambda-calculs Typés*. Thèse d'État, University of Paris VII, 1979.
- [Berry *et al.*, 1985] G. Berry, P.-L. Curien, and J.-J. Levy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 89–132. Cambridge University Press, 1985.
- [Breazu-Tannen *et al.*, 1990] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In M. Wand, editor, *Lisp and Functional Programming*, pages 44–60. ACM, 1990.
- [Bruce and Longo, 1990] K. B. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990.
- [Burn *et al.*, 1986] G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

- [Cardelli, 1987] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.
- [Cardelli, 1988] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
- [Clément *et al.*, 1986] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: mini-ML. In *Symposium on LISP and Functional Programming*, pages 13–27. ACM, 1986.
- [Damas, 1985] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Edinburgh University, 1985.
- [Damas and Milner, 1982] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, pages 207–212. ACM, 1982.
- [Duba *et al.*, 1991] B. Duba, R. Harper, and D. B. MacQueen. Typing first-class continuations in ML. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254. ACM, 1991.
- [Felleisen and Friedman, 1986] M. Felleisen and D. P. Friedman. Control operators, the secd-machine, and the  $\lambda$ -calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
- [Friedman, 1975] H. Friedman. Equality between functionals. In R. Parikh, editor, *Proceedings of the Logic Colloquium '73*, pages 22–37. *Lecture Notes in Mathematics* vol. 453, Springer-Verlag, 1975.
- [Girard, 1972] J. Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse d'État, University of Paris VII, 1972.
- [Gunter, 1992] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press, 1992.
- [Gunter, 1993] C. A. Gunter. Forms of semantic specification. In B. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science: Essays and Tutorials*, pages 332–353. World Scientific Publishers, 1993.
- [Gunter and Mitchell, 1994] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.
- [Gunter and Scott, 1990] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674. North-Holland, 1990.
- [Harper and Mitchell, 1993] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 1993. To appear.

- [Hindley and Seldin, 1986] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and  $\lambda$ -calculus*. Cambridge University Press, 1986.
- [Howard, 1973] W. Howard. Hereditarily majorizable functionals of finite type. In A. S. Troelstra, editor, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, pages 454–461. *Lecture Notes in Mathematics* vol 344, Springer-Verlag, 1973.
- [IEE, 1991] IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*, IEEE standard 1178-1990 edition, 1991.
- [Jim and Meyer, 1991] T. Jim and A. R. Meyer. Full abstraction and the context lemma. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 131–151. Springer-Verlag, September 1991.
- [Kahn, 1987] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.
- [Leroy, 1993] X. Leroy. Polymorphism by name for references and continuations. In S. L. Graham, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231. ACM, 1993.
- [MacQueen *et al.*, 1986] D. B. MacQueen, G. D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.
- [Martin-Löf, 1971] Per Martin-Löf. An intuitionistic theory of types. unpublished, 1971.
- [Meyer, 1982] A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.
- [Milner, 1978] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [Milner and Tofte, 1991] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, 1991.
- [Milner *et al.*, 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [Mitchell, 1990] J. C. Mitchell. Types systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 365–458. North-Holland, 1990.
- [Moggi, 1991] E. Moggi. A category-theoretic account of program modules. *Mathematical Structures in Computer Science*, 1:103–139, 1991.
- [Palsberg and Schwartzback, 1993] J. Palsberg and M. Schwartzback. *Object-Oriented Type Systems*. Wiley, 1993.
- [Plotkin, 1976] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5:452–487, 1976.



- [Plotkin, 1981] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Ny Munkegade—DK 8000 Aarhus C—Denmark, 1981.
- [Poigné, 1992] A. Poigné. Basic category theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science, Volume 1 Background: Mathematical Structures*, pages 413–640. Oxford University Press, 1992.
- [Reynolds, 1974] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425. *Lecture Notes in Computer Science vol. 19*, Springer, 1974.
- [Reynolds, 1980] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–258. *Lecture Notes in Computer Science vol. 94*, Springer, 1980.
- [Reynolds, 1983] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers (North-Holland).
- [Reynolds, 1984] J. C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, pages 145–156. *Lecture Notes in Computer Science vol. 173*, Springer, 1984.
- [Reynolds and Plotkin, 1990] J. C. Reynolds and G. D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, University of Texas at Austin Year of Programming, pages 127–152. Addison-Wesley, Reading, Massachusetts, 1990.
- [Robinson, 1965] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
- [Scedrov, 1990] A. Scedrov. A guide to polymorphic types. In P. Odifreddi, editor, *Logic and Computer Science*, pages 387–420. Academic Press, 1990.
- [Scott, 1969] D. S. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Unpublished manuscript, 1969, 1969.
- [Scott, 1976] D. S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, 1976.
- [Scott, 1981] D. S. Scott. Some ordered sets in computer science. In I. Rival, editor, *Ordered Sets*, pages 677–718. D. Reidel, 1981.
- [Scott, 1982a] D. S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *International Colloquium on Automata, Languages and Programs*, pages 577–613. *Lecture Notes in Computer Science vol. 140*, Springer, 1982.
- [Scott, 1982b] D. S. Scott. Lectures on a mathematical theory of compu-



- tation. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292. NATO Advanced Study Institutes Series, D. Reidel, 1982.
- [Sokolowksi, 1991] S. Sokolowksi. *Applicative High Order Programming: The standard ML perspective*. Chapman and Hall, 1991.
- [Statman, 1982] R. Statman. Completeness, invariance and  $\lambda$ -definability. *Journal of Symbolic Logic*, 47:17–26, 1982.
- [Statman, 1985a] R. Statman. Equality between functionals. In *Harvey Friedman's Research on the Foundations of Mathematics*, pages 331–338. North-Holland, 1985.
- [Statman, 1985b] R. Statman. Logical relations and the typed  $\lambda$ -calculus. *Information and Control*, 65:85–97, 1985.
- [Statman, 1986] R. Statman. On translating lambda terms into combinators: the basis problem. In A. Meyer, editor, *Symposium on Logic in Computer Science*, pages 378–382. ACM, 1986.
- [Stoughton, 1991] A. Stoughton. Interdefinability of parallel operations in PCF. *Theoretical Computer Science*, 79:357–358, 1991.
- [Tait, 1967] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32:198–212, 1967.
- [Tennent, 1992] B. Tennent. Denotational semantics. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science*. Oxford University Press, 1992.
- [Tofte, 1990] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.
- [Wright, 1993] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report COMP TR93-200, Department of Computer, Rice University, 1993.



# Index

- abstract bases 24
  - approximable relation between 27
  - see also* bases
- abstract machine 444
  - for  $ML_1$  calculus 444, 445
- acceptors 279–82
- Ada programming language 403
- adjunctions 34–7
  - basic adjunction 113–14
  - in domain theory 34–5, 37
- admissible predicates 14, 80
- admissible relations 80–1
  - co-induction with 82–3
  - induction with 81–2
- algebraic bc-domains 124
- algebraic domains 17–20
  - examples 17
  - vs. continuous domains 22
- algebraic semantics 323–91
  - goal 331
  - meaning of term 324, 326
  - notation used 332
- algebraic theories 332–43
  - basic identities 342–3
  - as categories 336–7
  - definitions 332–6
  - examples 337–41
  - with iteration operators 348–74
  - notations 341–2
  - ordered theories 343–8
    - definitions 343–4
    - examples 344–8
  - pictorial interpretations 335–6, 342, 348–53
  - see also* iteration...; iterative...; Lawvere...; ordered...; rational algebraic theory
- ALGOL60 language 243, 245, 256
- Algol-like language 245–89
  - and acceptors 279–82
  - and coercions 261–8
  - and jumps 282–7
  - and lists 274–9
  - local variables used 269–70
  - product types and arrays 270–4
  - programming logic applied 256–60
    - effect of jumps 286–7
  - semantics 250–3
  - syntax 246–9
  - variables treated 279–82
- ALGOL W language 270
- algorithm  $W$  439–40
  - soundness property 440–1
  - typechecking by 443
- alpha rule 227, 240
- answer-domain adjunction
  - morphism 309
- applicative language 205–28
  - combined with imperative language 245–89
  - call-by-value approach 253–6
  - programming logic applied 256–60
  - semantics 250–3
  - syntax 246–9
  - see also* Algol-like language
- defined notation 214–16
- definitions 206–12
- domain theoretic semantics
  - used 231–7
- elementary properties 217–20

- function applications 206–12
- function definitions 212–14
- identifiers treated 210–11
- operational semantics 237–9
- programming logic used
  - 220–8
- semantics 210–12
- syntax rules 207–10
  - Abstraction rule 214
  - Addition rule 209–10
  - Local-definition rule 208, 209, 210
  - Negation rule 208, 209
- see also* explicitly-typed language
- approximations, in domain theory 4, 15–27, 39
- arrays 272–3
- assertions, and Boolean expressions
  - 196, 257
- assignment commands 185–7
  - axioms for 198, 258–60, 304
  - semantic equation for 280
- assignments, run-time safety for
  - 441–6
- axiomatic domain theory 79,
  - 154–5
- axiomatic semantics 171, 196
- base morphisms 334
- bases
  - in directed-complete partial orders 16
  - ideal completion of 25
  - as objects 24–7
- bc (bounded-complete) domains
  - 54, 426
  - PCF types modelled 426
- Bekić's rule 74
- beta ( $\beta$ ) rule 227, 240
- bifinite domains 56–9, 123
- bilimits, of domains 48, 49–50
- binary numerals, denotational semantics used 172–5
- black-box representations,
  - algebraic theories 335–6, 342, 348–53
- block expressions 288–9
  - semantics of 308–11
- BNF (Backus–Naur Formalism) notation 172
- Boolean expressions, and
  - assertions 196, 257
- bottom element [in order theory]
  - 8, 345
- bound variable naming convention
  - 409, 440, 447
- call-by-name evaluation strategy
  - 236, 249, 253, 254
  - in PCF 422, 424
- call-by-value evaluation strategy
  - 253–6, 397
  - in PCF++ 456, 457
- call-by-value fixed-point
  - combinator
  - ML version 401, 402
  - Scheme version 400
- Cartesian closed categories of
  - domains 5, 52–65
  - finite choice 55–62
  - hierarchy of categories of domains 62–5
  - local uniqueness 54–5
- categories, algebraic theories as
  - 336–7
- category theory
  - and domain theory 34, 39–40
  - and possible-worlds semantics 290, 291–2
- central limit–colimit theorem 47, 70
- CF<sub>G</sub>** subtheory 344, 363
- chains [in order theory] 10
- Chomsky normal form theorem 375
- closure operators 39
- closure system 9
- coalesced-sum calculations 43, 124

- coercions 247–8, 261–8
- cofinal subsets 11
- coherence
  - in applicative language 218
  - in domain theory 60–2, 119–20, 123, 126
- coherent algebraic domains 123
- coherent algebraic prelocale 126
- coherent domains 60–2
- coherent space 119–20
- co-induction theorem 82–3
- colimiting morphisms 45
- colimits 45, 48
- compact domains 55–62
- compact-open sets, and spectral spaces 120–1
- compact-open subsets
  - lattice of 123–4
  - domain construction using 124–9
- compact saturated sets 60–1
- compatible partial ordering 371
- complete lattices 9
  - points characterization 112–13
- completely prime filter 112
- completeness for sets 418–20
- completions 282–5
  - see also* program-completion functions
- compositionality [in imperative language] 175–6
- composition operations [in algebraic theories] 334, 339
- composition operator [in algebraic theories] 390
- computational domains, *see* domains
- computational monads 91, 153–4
- conditional variables 246–7
- connecting morphisms 45
- context-free grammar 331, 363–4
- continuation semantics 283
  - Hoare-triple form interpreted using 286
- continuations [for command execution] 283–4
- continuity 3–4
- continuous directed-complete partial order 18
- continuous domain-algebras 87–92
- continuous domains 17–20
  - examples 18
  - retract of 33
  - vs. algebraic domains 22
  - vs. directed-complete partial orders 22
- continuous embedding projection pairs (e-p-pairs) 37–8
- continuous functions 13–15, 232, 235–6
- continuous function space 13, 41–2
- continuous functors 70–1
- continuous idempotent functions 34
- continuous insertion closure pairs (i-c-pairs) 39
- continuous lattices, and locally compact spaces 118–19
- continuous section retraction pair 32, 33
- contravariant functors 294
- convergence, in domain theory 6–15
- convex hull 96
- convex powertheory 94–6
  - see also* Plotkin powerdomain
- counter objects, class defined 271–2
- covariant functors, restriction to embeddings 70–2
- $\mathbf{CT}_\Sigma$  [algebraic] theory 345, 355, 358, 379
- Currying 228, 399
- data-type pre-orders 262



- data types 68–9, 180, 206
  - datatypes, recursive definitions 2
  - dcpos (directed-complete partial orders) 12
    - adding new bottom element 44
    - bases in 16
    - Cartesian product of 40–1
    - dcpo-semilattice 94
    - examples 12
    - free dcpo-algebras 85–6
    - Scott-topology on 28–9
  - declarative language, *see* applicative language
  - del-operator 356
  - denotational semantics 169–311
    - and domain theory 4, 46, 107, 147
    - example 172–5
    - exercises covering 175, 190–1, 203, 223, 225, 239, 260, 273, 278, 286
    - meaning of term 171
    - and programming languages 178
      - Algol-like language 245–89
      - applicative language 205–28
      - imperative language 180–205
  - dI domains 427
    - PCF types modelled 427
  - directed-complete partial orders 12
    - see also* dcpo
  - directed sets 10–12
    - vs.  $\omega$ -chains 20–1
  - disjoint union 42
  - distinguished morphisms 334, 339, 341
  - distributive lattices 118
    - spectrum of 121
  - divergence 431
  - domain-algebras 87–92
  - domain logic 145–7
  - domain prelocales 126–8, 140–1
    - see also* sub-prelocales
  - domains
    - Cartesian closed categories 5, 52–65
    - collectively 5, 32–50
    - comparing of 32–9
    - construction of
      - finitary constructions 39–44
      - function-space construction 133–6
      - infinitary constructions 44–50
    - languages used for points/properties/types 141–7
    - and Plotkin powerlocale 136–9
    - and recursive domain equations 139–41
    - step-by-step technique 129–33
    - working with lattices of compact-open subsets 124–9
    - see also* recursive domain equations
  - expanding sequence of 46–7
  - hierarchy of categories 62–5
  - individually 6–32
  - Scott-topology on 29
  - simple types as 420–30
  - as types 3
  - with least element 62–4
  - without least element 64–5
  - see also* algebraic...; bifinite...; coherent...; compact...; continuous...; effectively given...; flat...; FS...; lattice like...; powerdomains; pre-domains; quotient...; universal domains
- domains of interpretation 172

- domain-theoretic semantics 150, 231–7
- domain theory 1–157
  - applications 156–7
  - approximations in 4, 15–27
  - and convergence 6–15
  - and equational theories 5–6, 84–105
  - exercises covering 30–2, 50–2, 65–6, 83–4, 105–7, 147–8
  - further topics 148–56
  - literature 156–7
  - and logic 6, 107–47
  - reformulations of 152–4
  - and topology 27–30
  - see also* axiomatic domain theory; recursive domain equations; synthetic domain theory
- dynamic storage 268
  - semantics of 269–70
- effectively given domains 148–9
- Egli–Milner ordering 101, 102, 103, 136
- Egli–Milner relation 95, 97
  - see also* topological Egli–Milner ordering
- Elgot-theories 356
- empty string 333
- environments
  - in applicative language 205–6
  - classification by type assignments 413
- e-p (embedding projection) pairs 37–8
- equational judgements 411
- equational theories 5–6, 84–105, 411
  - general techniques 85–94
  - free continuous domain algebras 87–92
  - free dcpo-algebras 85–6
  - least elements and strict algebras 92–4
  - and powerdomains 94–105
- erasure of terms
  - in polymorphic  $\lambda$ -calculus 468–9
  - in simply-typed  $\lambda$ -calculus 447
- eta ( $\eta$ ) rule 227, 240, 303, 414, 447, 453
- expanding sequences [of domains] 46–7, 59
- explicitly-typed  $\lambda$ -calculus 436
  - see also* simply-typed  $\lambda$ -calculus
- explicitly-typed language 217, 264
- expressible meanings of types 255
- expression continuations 289
- extended type assignment 417
- extensional environment model 415
- Extensionality law 227, 240
- F*-algebras 68–9, 84–5
- filtered sets 10
- finite amalgams 64–5
- finite mub property 56
- fixed-point combinator 244
- fixed-point equations 230
- fixed-point induction 14, 237
- fixed-point induction axiom 241, 242
- fixpoints 2
- flat domains 17
- flat natural numbers 7
- flowcharts
  - example 326–30
  - interpretations 364–6
  - and iteration operators 356–64
  - representation 338
- Floyd–Hoare logic 196
  - axioms 201
  - syntax and semantics of language 196–7
  - see also* programming logics

- formal union [operation] 94
- frame distributivity law 111
- frame-homomorphism 111
- frames 111, 116
- free continuous domain-algebras 87–92
- free dcpo-algebras 85–6
- free iteration  $\Sigma$ -(generated-) theory 380
  - see also*  $\mathbf{I}_\Sigma$  theory
- free  $\omega$ -continuous 1-sorted  $\Sigma$ -recursion theory 346–7, 388
  - see also*  $\mathbf{Rec}_\Sigma$  theory
- free  $\omega$ -continuous  $S$ -sorted  $\Sigma$ -(generated-)theory 345
  - see also*  $\mathbf{CT}_\Sigma$  theory
- free ordered  $\Sigma$ -(generated-)theory 346, 380
  - see also*  $\mathbf{FT}_\Sigma$  theory
- free  $S$ -sorted rational  $\Sigma$ -(generated-)theory 379–80
  - see also*  $\mathbf{RT}_\Sigma$  theory
- free  $S$ -sorted  $\Sigma$ -(generated-)theory 339
  - see also*  $\mathbf{T}_\Sigma$  theory
- free theories 379–80
- FS-domains 59–60
- $\mathbf{FT}_\Sigma$  [algebraic] theory 346, 380
- full abstraction of semantic interpretation 177, 242–3
  - and PCF system 427–30
  - practical significance 177–8
- functional languages
  - lists in 274
  - see also* applicative language
- functional types 206–7
  - coercions on 263
- function definitions, in applicative language 212–14
- function space 13, 41–2
  - construction of 133–6
- function space operator 407–8
- functor–category semantics 291–2
- generic [in Ada language] 403
- genuine equations, as example of recursive domain equations 67
- Girard–Reynolds polymorphic  $\lambda$ -calculus 404, 458, 461–4
  - equational rules 464
  - typing rules 463
- Girard’s System F 150
- Hasse diagrams 7, 262, 278, 281, 383
- Hausdorff topology 18, 28, 60, 62
- $H$ -environments 413, 416, 452, 467
- Herbrand interpretations 379, 380–5
- higher-order functions, use in programming 397–400
- Hoare-double specifications 286
- Hoare powerdomain 96, 97, 98
  - topological representation of 98–9
- Hoare powerlocale 139
- Hoare-triple specifications 196, 197, 258
- Hofmann–Mislove theorem 60, 104, 116–17
- hulls 9, 96
- hyperspaces 104
- i-c (insertion closure) pairs 39
- ideal morphisms 356
- idempotent deflations 58
- idempotents 33–4
- idempotent self-maps 34
- image domains 53, 54–5
- imperative language 180–205
  - assignment commands 185–7
  - combined with applicative language 245–89
  - call-by-value approach 253–6

programming logic applied  
     256–60  
 semantics 250–3  
 syntax 246–9  
     *see also* Algol-like language  
 command equivalences 185  
 commands as programs  
     191–2  
 expressions and commands  
     180–5  
 indefinite iterations 187–91  
 non-determinism 203–4  
 operational semantics 192–6  
     trace of computation 193,  
     194  
 programming logic used 196–  
     203  
 semantics 183–5  
 syntax 180–3  
 variable-identifier 186  
 Implication-Introduction rule 242  
 implicitly-typed  $\lambda$ -calculus 434  
 implicit types 434–41  
 impredicativity 466  
 inclusive subsets, as types 451–5  
 indefinite iterations 187–90  
 inductive closure 17, 385  
 inductive interpretations 385  
 inequations 383  
 inference rules 172, 173  
 infimum [in order theory] 9  
 information systems 108, 150–1  
 initial  $F$ -algebras 68–9  
 injective functions 267  
 insertion closure pairs 39  
 intermediate output 287–8  
 intuitionistic logic 199, 303  
 invariants, types as 430–46  
 irreducible closed set 114  
 iteration closure 375  
 iteration operators 348–9  
     and flowcharts 356–64  
     identities for 372–4  
 iteration theories 349, 350–4

Herbrand interpretation 381  
     relationship with iterative and  
     rational theories 366–72  
 iterative theories 349, 355–6  
     relationship with iteration and  
     rational theories 366–72  
 joinable subsets 56, 57  
 join-approximable relations 121  
 jumps, in Algol-like language  
     282–7  
 kernel operators 38  
     with finite images 58  
 Kleisli category construction 386  
 Kleisli extension 153  
 lambda calculus  
     logical principles preserved in  
     [sc ALGOL]60 language  
     256  
     *see also* simply-typed...; typed  
     ...; untyped  $\lambda$  calculus  
 lambda expression 213  
     semantic equation for 214  
 lattice-like domains 54–5  
 Lawson-topology 62, 103  
 Lawvere algebraic theory 331, 332  
 lazy natural numbers 7  
 LCF (Logic for Computable  
     Functions) system 239,  
     245  
 L-domains 54–5  
 least fixed points, iteration  
     operator defined in terms  
     of 354  
 least-fixed-point theorem 231–2  
 least fixpoint operators 14  
     invariance of 14–15, 75, 76  
 least fixpoints 2, 13  
 least homomorphisms 74, 75  
 least substitutive preorder 88  
 least upper bounds 264, 265  
     *see also* mub (minimal upper  
     bound)...

- lifting [operation] 44, 153, 275, 454
- limit-colimit coincidence 46–9
- limiting morphisms 45
- linear logic 154
- linear types 154
- lists, in Algol-like language 274–9
- locally compact spaces 118–19
- locally continuous functors 71–2
- local-variable declarations 298, 305
- local variables
  - in Algol-like language 268–70
  - semantics of 298–301
- logic, and domain theory 6, 107–47
- logical relations 239, 420
- lower sets 8
- many-sorted Lawvere algebraic theories
  - definitions 332–6
  - see also* algebraic theories; Lawvere algebraic theory
- maximal elements 8
- $M$ -construction 386–91
- metalanguage 174
  - see also* ML language
- metavariables 172
- Mezei–Wright theorem 364
- Milner’s algorithm  $W$  439
  - see also* algorithm  $W$
- minimal elements 8
- minimalization operator [in algebraic theories] 390
- minimal upper bound, *see* mub...
- mixed endomorphisms 78
- mixed-variant functors 72, 77–9
- $ML_0$  calculus
  - assignments in 437
  - with  $\Pi$  introduction and elimination 461
  - types 436
    - alternative typing system 460–1
    - sets as model of 458–60
    - typing rules 437, 438, 461
- $ML_1$  calculus
  - abstract machine for 444, 445
  - grammar 444
- ML language 270, 397
  - polymorphic swapping in 403
  - type-checking 399, 401, 404
- MODULA2 language 270
- monads 91, 153–4
- monotone nets 11
- monotone [order-preserving] functions 9–10
- monotone section retraction pair 32
- Monotonicity law 240, 265
- mub (minimal upper bound)
  - closure 56, 57
- mub (minimal upper bound) completeness 56
- multiple definitions, notation used in applicative language 215–16
- natural [operational] semantics 423
  - for PCF system 423–4
- non-bifinite domains 57
- non-continuous directed-complete partial order 18
- non-deterministic control structures 203–4
- non-interference specifications 304–8
- normal form theorems 375–8
- object-oriented programming, types used 271–3, 470
- $\omega$ -chains 10, 343, 384
  - vs. directed sets 20–1
- $\omega$ -continuous ordered algebraic theories 343, 345
  - Herbrand interpretation 383–4
- open sets 27



- operational equivalence, for PCF system 425–6
- operational semantics 171
  - for applicative language 237–9
  - for imperative language 192–6
  - for PCF systems 422–5, 432, 456–7
- order of approximation 15–16
- ordered algebraic theories 343–8
  - definition 343–4
  - examples 344–8
- order-preserving mapping 10
- order relation 82
- order theory 7–8
  - notation 8–9
- pairing operation [in algebraic theories] 341
- parameter passing
  - call-by-name evaluation 236, 249, 253, 254
  - call-by-value evaluation 254–6
- parametric polymorphism 402–4
- Park's theorem 242
- partial continuous functions 152–3
- partial correctness 197
- partial equivalence relations
  - definition 466
  - as model of polymorphic types 468–70
  - as model of subtypes 470
  - simple types as 466–8
  - types as 458–70
- partial homomorphisms 419–20
- partial information, and domain theory 4
- partially ordered sets 6–7
  - see also* posets
- PASCAL language 246, 269, 270, 273
- path compression 311
- PCF (programming language for computable functions) 178, 420–1
  - evaluation context for 423
  - fully abstract model sought 427–30
  - natural rules for call-by-name evaluation 424
  - operational equivalence 425–6
  - operational semantics 422–5
    - plus records and variants (PCF+) 455
    - typing rules 455, 456
  - plus records and variants and subtyping (PCF++) 455
    - rules for call-by-value evaluation 457
    - rules for subtyping 456
  - syntax 421
  - transition rules for call-by-name evaluation 422
  - with type errors 432, 454
  - typing rules 421
  - see also* simply-typed  $\lambda$ -calculus
- PERs *see* partial equivalence relations
- phrase assignments 222
- phrase types 172, 180, 206
- Plotkin-order 56, 57
- Plotkin powerdomain 94–6
  - topological representation of 100–2
- Plotkin powerlocale 136–9
- pointed directed-complete partial orders
  - coalesced sum of 43
  - strict function between 13
- pointed posets 8, 12
- points [of domains], language used 144–5

- polymorphic  $\lambda$ -calculus 404, 458, 461–4
  - equational rules 464
  - typing rules 463
  - see also* Girard–Reynolds...
- polymorphic types
  - partial equivalence relations
    - as model of 468–70
  - sets as model of 464–6
- polymorphism
  - in domain-theoretic semantics 150
  - meaning of term 403
  - see also* parametric polymorphism
- posets 6–7
  - attribute-classified 405
  - drawn as line diagrams 7
- possible-worlds semantics 290–311
  - category-theoretical
    - formulation 291–2
  - local variables 298–301
  - non-interference specifications 304–8
  - specifications 302–3
- Pow<sub>A</sub>** [algebraic] theory 340
- powerdomains
  - Hoare powerdomain 96, 97, 98
  - one-sided powerdomains 96–8
  - Plotkin powerdomain 94–6
  - probabilistic version 104–5
  - Smyth powerdomain 96, 97, 98
  - topological representation of 98–104
  - see also* Hoare...; Plotkin...; Smyth powerdomain
- predicate transformers 205
- predomains 152
- pre-frame 415, 417
- pre-isomorphism 126
- prelocales 126
- preorderance 381, 382
- preorders 7, 262
- principal types 441
- procedural languages 245
  - see also* Algol-like language
- procedural types, coercions on 263
- product of morphisms 342–3
- product types 270–1
  - use in object-oriented programming 271–3
- program-completion functions 191, 282–5
- programming language for
  - computable functions 178, 420–1
  - see also* PCF
- programming logics
  - for Algol-like language 256–60
  - axioms 258–60
  - effect of jumps 286–7
  - syntax and semantics 257–8
  - for applicative language 220–8
  - augmented by recursion 239–42
  - axioms 226–8, 240–2
  - and substitutions 221–6
  - syntax and semantics 220–1, 240
  - and domain theory 107
  - for imperative language 196–203
  - axioms and rules 197–201
  - soundness considerations 201–3
  - syntax and semantics 196–7
- projections 38
- properties, language used in domain theory 142–4
- quantification notation, as
  - example of defined nota-

- tion in applicative language 216
- quotient domains 39
- rational closure theorem 379–80
  - see also*  $\mathbf{RT}_\Sigma$  theory
- rational theories 349, 354–5
  - Herbrand interpretation 381–3
  - relationship with iteration/iterative theories 366–72
- records 405–6
  - fields of 405–6
  - PCF extended to include 455–6
  - rules [in PCF++] for call-by-value evaluation 457
- $\mathbf{Rec}_\Sigma$  [algebraic] theory 346, 355, 360, 361, 365, 388
- recursion equations, example 330–1
- recursive definitions 67–8, 228–31
- recursive domain equations 2–3, 5, 66–83, 139–41
  - analysis of solutions 79–83
    - and admissible relations 80–1
  - co-induction with admissible relations 82–3
  - induction with admissible relations 81–2
  - structural induction on terms 79–80
- canonicity 74–9
  - and finality 77
  - and initiality 76–7
  - and invariance 75–6, 78
  - and minimality 76, 79
  - and mixed variance 77–9
- construction of solutions 69–74
  - continuous functors 70–1
  - for lists 275–9
  - locally continuous functors 71–2
  - parameterized equations 72–4
  - examples 67–9
    - data types 68–9
    - genuine equations 67
    - recursive definitions 67–8
- recursive function definition 228–9
- recursive hierarchies 386–91
- recursive types 400–2, 470
- Reflexivity law 226, 240
- retractions 32–3
- R-structures 24
  - see also* abstract bases
- $\mathbf{RT}_\Sigma$  [algebraic] theory 379–80
- run-time safety 430–4
  - for assignments and continuations 441–6
- run-time type errors 398
- Russell's paradox 465–6
- Scheme programming language 397
  - type checking 401, 404
- schizophrenic object 113
- Scott-closure [of sets] 28
- Scott-continuous functions 13–15
- Scott-continuous homomorphisms 86
- Scott-domain 54
- Scott-topology
  - on directed-complete partial orders 28–9
  - on domains 29
- section retraction pairs 32–3
- semantic-domain functors 292–5, 302
- semantic equations 174
  - for applicative language 211, 214
  - for call-by-value based language 255

- for imperative language 184, 197
- semantics
  - for Algol-like language 250–3
  - for applicative language 210–12
  - for binary-numeral language 172–4
  - of block expressions 308–11
  - criteria for 176–8
  - for imperative language 183–5
  - meaning of term 170
  - for programming logics 196–7, 220–1, 240, 257–8
  - see also* algebraic...; axiomatic...; denotational...; domain-theoretic...; functor-category...; operational...; possible-worlds semantics
- semantic valuations 295–8
- semilattices 9
- sequential functions 151
- sets
  - as model 412–15
  - as model of  $ML_0$  types 458–60
  - simple types as 408–20
- SFP 59
- $\Sigma$ -flowcharts 378
- simple types
  - as domains 420–30
  - as partial equivalence relations 466–8
  - as sets 408–20
- simply-typed  $\lambda$ -calculus 408–9
  - equational rules 412
  - equations 411
  - semantics 413–14
  - soundness property 414
  - type assignment 409–10
  - type frames as models 415
  - typing rules 410
  - see also* PCF
- SIMULA language 270
- smash product 43
- Smyth powerdomain 96, 97, 98
  - topological representation of 99–100
- Smyth powerlocale 139
- sober spaces 18
  - properties 116–18
  - and spatial lattices 114–16
- SOS (structural operational semantics) 422
- spatial lattices 115
- specialization order 117
- spectral spaces 120–2
- $S$ -sorted algebraic theory 333–4
  - as category 336
- $S$ -sorted signature 338, 344
- stable ordering 151
- stability modulus 151
- state-change restriction morphism 307, 308, 309
- state-set restriction morphism 307
- step functions 52–3, 56
- Stone duality 108–14
  - in domains construction 131
  - overview of Stone-dualities in domain theory 125
- strict algebras 92–4
- strict function space 13, 43
- strict homomorphism 93
- structural induction 175, 194, 218, 224, 225, 265
- structural operational semantics 422
- structured types 270
- sub-domains 39, 278
- subject reduction theorem 430, 433
- sub-prelocale 128, 129
- subsets
  - inclusive subsets as types 451–5
  - types as 446–58
- substitutions

- in  $ML_0$  calculus 436, 437, 438, 439
- in programming logic for applicative language 221–6
- Substitutivity law 226, 240
- subsumption rule, added to PCF+ 456
- subtypes 404–8, 455
  - partial equivalence relations as model of 470
- subtyping
  - PCF+ extended to include 455–6
  - as subset inclusion 455–8
- Sum<sub>A</sub>** [algebraic] theory 337, 355
- supremum [in order theory] 8–9
- symmetry axiom 226
- syntactic translation 448
- syntax
  - Algol-like language 246–9
  - applicative language 208
  - binary-numeral language 172, 173
  - imperative language 180–3
  - programming logics 196–7, 220, 240, 257
- synthetic domain theory 155–6
- System F 150
- term algebras 68
- term models 417, 418
- theory morphisms 334, 339
  - and flowcharts 358
- ticking programs 288
- topological Egli–Milner ordering 101, 102, 103
- topology
  - and domain theory 27–30
  - see also* Hausdorff...; Lawson...; Scott-topology
- transitive relations, order theory as study of 7
- Transitivity law 226, 240
- T<sub>Σ</sub>** [algebraic] theory 339, 346
- TT<sub>Σ</sub>** [algebraic] theory 346, 372
- tupling operations 334, 339
- turnstile symbol 411
- 2/3-binite domains 123
- 2/3-SFP Theorem 61
- type abstraction 462
- type application 462
- type assignment 409–10
- type-checking 400–1, 403–4
- typed  $\lambda$  calculus
  - programming language based on 397
  - see also* ML language
- type errors, operational rules for 432
- type expressions, language used in domain theory 141–2
- type frames 415–18
- type inference 403, 441
- types
  - applicative language 206–7
  - imperative language 180–1
  - inclusive subsets as 451–5
  - as invariants 430–46
  - as partial equivalence relations 458–70
  - in programming 397–408
    - advantages of use 396–7
    - semantics of 395–471
    - as subsets 446–58
    - see also* implicit...; principal...; recursive...; simple types
- type theories, and universal domains 150
- type-value environment 459, 464
- type variable 402
- typing derivation 410
- typing judgement 410, 437
- unifiers 439
- universal domains 149–50, 451
  - and type theories 150



untyped  $\lambda\beta$ -calculus, equational  
rules 447

untyped  $\lambda\beta\eta$ -calculus 447

untyped  $\lambda$ -calculus 244, 446–8

model for 448–51

programming language based  
on 397

*see also* Scheme language

untyped procedures 243–5

upper sets 8

valuation functions 171, 173, 233

variants 406

fields of 406

PCF extended to include  
455–6

rules [in PCF++] for call-by-  
value evaluation 457

weak upper topology 117

**while** loop, invariant assertion of  
198, 199

**while** programs

language used 205

*see also* imperative language

write-only variables 279

xi ( $\xi$ ) rule 453



















3 9001 03441 2208





0 19 853/32X